CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
COORDENAÇÃO DO CURSO TÉCNICO EM ELETRÔNICA

Felipe Bordoni Diniz

LAUNCHPAD F28377S

Volume 1: ADC, PWM e DAC

Belo Horizonte - MG
2018

Felipe Bordoni Diniz

LAUNCHPAD F28377S

Volume 1: ADC, PWM e DAC

Belo Horizonte - MG
2018

Resumo

O *DSP* TMS320F28377S produzido pela *Texas Instruments*™ é um poderoso dispositivo para aplicações em controle de processos em malha fechada devido a seus aceleradores de processamento, que conseguem efetuar operações em um número inferior de ciclos do que um microcontrolador normal.

Neste tutorial, é visada uma introdução para a utilização deste dispositivo, de forma a fornecer ao seu leitor uma maior velocidade para familiarização com o *DSP*, para que o mesmo possa conseguir desenvolver seu código de aplicação com menos empecilhos.

Primeiramente será dada uma introdução a *IDE* de desenvolvimento da *Texas Instuments*™, o *Code Composer.* Em seguida, serão apresentadas informações e explicações acerca dos módulos periféricos *ADC* (*Analog Digital Converter*) e *ePWM* (*Enhanced Pulse Width Modulator*), apresentando códigos de exemplo comentados, de forma a proporcionar ao leitor uma rápida assimilação de comandos e inicialização destes módulos.

Sumário

LISTA DE FIGURAS **6**

1. INTRODUÇÃO **7**

2. CARACTERÍSTICAS DO *DSP* **8**

2.1 AMBIENTE DE DESENVOLVIMENTO 9

2.2 MODELOS DE CODIFICAÇÃO 10

2.3 IMPORTANDO E CRIANDO UM PROJETO 10

3. INICIALIZAÇÃO DO *DSP* E *GPIOS* **13**

3.1 INICIALIZAÇÃO 13

3.2 DEFINIÇÃO DE *GPIOS* 14

4. CONVERSOR ANALÓGICO-DIGITAL (*ADC*) **16**

4.1 INTRODUÇÃO 16

4.2 PRINCÍPIO DE AMOSTRAGEM E CONVERSÃO 16

4.3 START-OF-CONVERSION 17

4.4 INICIALIZAÇÃO DO *ADC* 19

4.5 CARACTERIZAÇÃO DO *ADC* 19

4.6 INTERRUPÇÕES DO *ADC* 20

4.7 PRIORIDADE DE CONVERSÃO (*ROUND ROBIN*) 21

4.8 EXEMPLO 1 – CONVERSÃO CONTÍNUA 24

4.9 EXEMPLO 2 – CONVERSÃO COM *SOC* VIA *SOFTWARE* 29

5. MÓDULO *EPWM* **32**

5.1 INTRODUÇÃO 32

5.2 CANAIS *EPWM* 33

5.3 MODOS DE CONTAGEM E BASE DE TEMPO (*TB*) 34

5.4 TEMPO-MORTO (*DB*) 36

5.5 *ACTION-QUALIFIER* (*AQ*) 38

5.6 *COUNTER-COMPARE* (*CC*) 39

5.7 *EVENT-TRIGGER* (*ET*) 40

5.8 EXEMPLO – *ADC* COM *SOC* DADO PELO *EPWM* 41

6. *BUFFERED DIGITAL-TO-ANALOG CONVERTER* (*DAC*) **48**

6.1 INTRODUÇÃO 48

6.2 FUNCIONAMENTO 48

6.3 *DAC* EM OPERAÇÃO.....49
6.4 BIBLIOTECA *SGEN*.....50
6.5 EXEMPLO – GERAÇÃO DE UM SINAL CC.....52

7. CONCLUSÃO.....55

REFERÊNCIAS BIBLIOGRÁFICAS.....56

APÊNDICE A.....58

APÊNDICE B.....59

APÊNDICE C.....62

APÊNDICE D.....64

APÊNDICE E.....68

APÊNDICE F.....75

APÊNDICE G.....76

Lista de Figuras

FIGURA 2.1: ESQUEMÁTICO DA *LAUNCHPAD™C2000™DELFINO™*TMS320F28377S.....9

FIGURA 2.2:DIRETÓRIO PARA PROJETOS EXEMPLOS DO TMS320F28377S.....11

FIGURA 4.1: CIRCUITO DE *S&H* EM 12 *BITS*.....17

FIGURA 4.2: CIRCUITO DE *S&H* EM 16 *BITS*.....17

FIGURA 4.3: DIAGRAMA DE BLOCOS PARA OS *SOCS*.....18

FIGURA 4.4: EXEMPLO DE *ROUND ROBIN*.....22

FIGURA 4.5: EXEMPLO DE *ROUND ROBIN* EM ALTA PRIORIDADE.....24

FIGURA 5.1: ONDA *PWM*.....32

FIGURA 5.2: DIAGRAMA EM BLOCOS DE UM MÓDULO *EPWM*.....33

FIGURA 5.3: CONTAGEM DECRESCENTE DO *EPWM*.....35

FIGURA 5.4: CONTAGEM CRESCENTE E DECRESCENTE DO *EPWM*.....35

FIGURA 5.5: TEMPO-MORTO NA SAÍDA *EPWM*.....37

FIGURA 5.6: MUDANÇA NO CICLO DE TRABALHO.....40

FIGURA 5.7: CONEXÕES DOS *BITS* DO MÓDULO *EVENT-TRIGGER*.....41

FIGURA 6.1: IMPLEMENTAÇÃO DO *DAC* COM FILTRO.....48

FIGURA 6.2: DIAGRAMA DE BLOCOS DO *DAC*.....49

FIGURA 6.3: LINEARIDADE DO *DAC*.....50

FIGURA 6.4: BLOCO SIMPLIFICADO DA *RMPGEN*.....51

1. Introdução

Em controle de processos via dispositivos digitais, a velocidade de aquisição de dados é uma característica vital para a qualidade dos resultados e da ação de controle ao processo a ser controlado. Os microcontroladores em geral podem suprir a velocidade de processamento necessária para pequenos processos, em que o desempenho não precisa necessariamente ser tratado como algo vital, porém quando o processamento é uma necessidade dominante, é necessário que sejam utilizados dispositivos que possuam uma capacidade maior de processamento.

Os *DSPs* (*Digital Signal Processor*) apresentam diversos aceleradores de processamento internos, que trabalham em paralelo com a *CPU* principal, permitindo que operações, e aquisição de dados possam ser feitos com uma maior velocidade. Ao se utilizar um *DSP*, o usuário tem o entendimento de que a aplicação do dispositivo tem justificativa baseada na virtude da velocidade em processamento que o mesmo oferece ao processo a ser controlado, como por exemplo, processamento de áudio e vídeo, controle de motores, reconhecimento de íris, dentre outros.

Neste tutorial abordaremos o *DSP* da série *C2000™Delfino™*TMS320F28377S constituído pelo núcleo de processador *C28x™* trabalhada na plataforma *Launchpad™* produzido pela *Texas Instruments™.* Esta plataforma apresenta a capacidade de trabalhar em aplicações em tempo real graças à capacidade de aquisição de dados e rápido processamento de sinais, que como já dito anteriormente, é um trunfo que um *DSP* possui em comparação à microcontroladores comuns.

À diante será apresentada uma introdução à utilização do software para aplicações de desejo do usuário. Os tópicos apresentarão primeiro uma visão acerca das características do F28377S, ao ambiente de desenvolvimento de código *Code Composer Studio*, e os modelos de programação. Após isto serão apresentados os periféricos *ADC* (*Analog Digital Converter*) e *ePWM* (*Enhanced Pulse Width Modulator*), explicando seu funcionamento e exibindo um código de exemplo comentado, visando uma assimilação mais confortável ao leitor.

2. Características do *DSP*

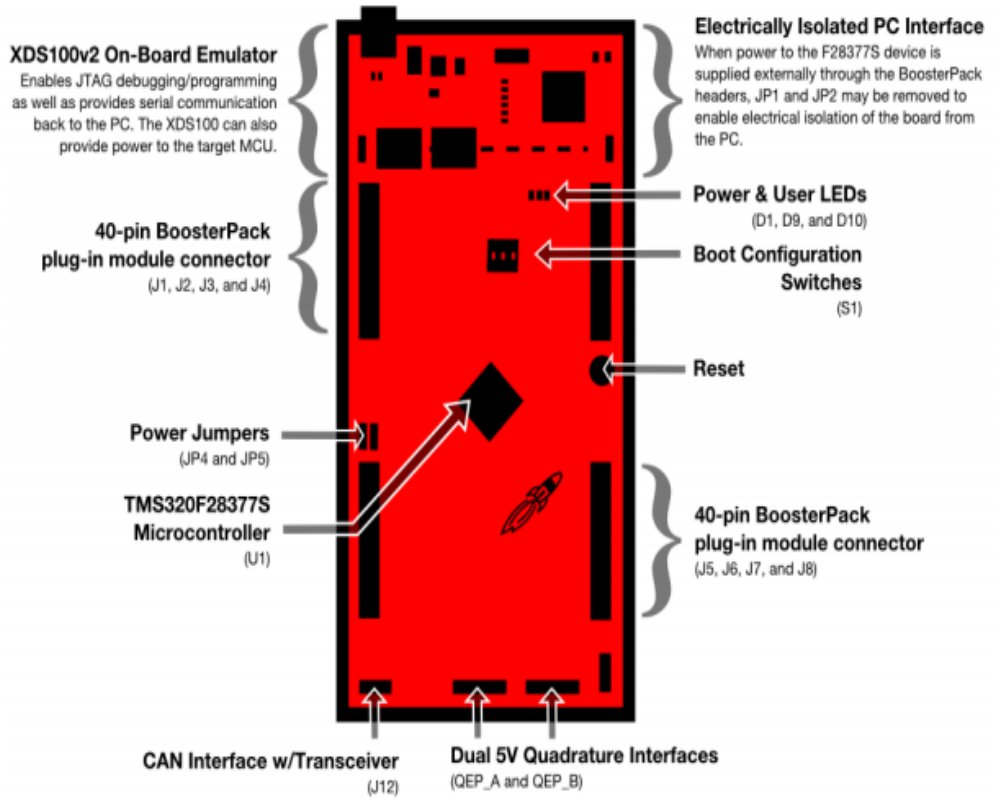
O *DSP* TMS320F28377S apresenta as seguintes especificações:

- Arquitetura de 32-*bits*;
- Processador C281 (Contém os aceleradores *FPU, VCU-II, TMU* tipo-0, *CLA* tipo-1);
- *Clock* de 200MHz;
- *ADC* de 16-*bits,* podendo trabalhar também como 12-*bits* (14 canais, podendo operar como *Differential* ou *Single Ended*);
- 15 canais *ePWM* do tipo 4;
- 9 canais *HRePWM*do tipo4;
- 3 *CPU Timers* de 32-*bits*;
- 41 pinos de *GPIO* (*General Purpose Input/Output*);
- Comunicação *I2C*, *SPI*, *UART*;
- 1 MB de memória *flash*;
- 164 kB de memória *flash* (36 kB dedicados e 128 kB compartilhados);

A plataforma *Launchpad*™já contém todos os *hardwares* e *softwares* (*ControlSUITE*™) que são necessários para que o F28377S possa operar em sua totalidade. A ferramenta *JTAG* é a responsável pela aliança entre o *DSP* e o computador onde o usuário desenvolverá seu código de aplicação. A interface de comunicação serial *UART* é responsável pela comunicação através da conexão *USB*.

A *Launchpad*™apresenta dois *LEDs*para aplicações do usuário, uma chave *Push-Bottom* para *reset*, uma chave para configurações de *boot*.A Figura 2 . 1apresenta uma ilustração esquemática que apresenta uma visão geral acerca da *Launchpad*™.

Figura 2.1: Esquemático da *Launchpad™C2000™Delfino*™TMS320F28377S.



Fonte: SPRUI25C. [1]

2.1 Ambiente de Desenvolvimento

O ambiente de desenvolvimento integrado (*IDE – Integrated Development Environment*) utilizado para a elaboração de códigos para aplicação é o *Code Composer Studio*™, desenvolvido pela própria *Texas Instruments*™. A versão utilizada da *IDE* é a 6.2.0.00050, disponibilizada gratuitamente no site da *Texas Instruments*™ [2]. Para realizar o *download* é necessário que o usuário crie uma conta no site e se registre.

Aliado ao *Code Composer Studio*™ foi utilizado o *software ControlSUITE*™, que constitui de um acervo de *drivers*, manuais, bibliotecas e programas exemplos de diversas plataformas de dispositivos desenvolvidos pela *Texas Instruments*™. Para os iniciantes, é uma excelente fonte de consulta acerca do dispositivo a ser trabalhado pelo usuário. O *ControlSUITE*™ também é disponibilizado no site da *Texas Instruments*™ gratuitamente, mediante cadastro no site [3].

2.2 Modelos de Codificação

A programação feita via *Code Composer*™ pode ser realizada via dois modelos de programação, *DRA* (*Direct Register Access Model*) e *SD* (*Software Driver Model*). Para este tutorial o modelo utilizado será o *DRA* devido a possibilitar uma melhor percepção acerca dos registros a serem escritos e lidos, e principalmente por termos a preocupação com relação ao tempo de processamento e resposta do *DSP* com relação aos processos com a aplicação do mesmo.

O modelo *DRA* trabalha em uma linguagem de menor nível de abstração do que a *SD*, necessitando assim, que o usuário ao realizar a programação, tenha conhecimentos acerca da estrutura de registradores do dispositivo. A linguagem utilizada para a programação destes dois modelos é a linguagem C.

Em questão de eficiência e tamanho de código, recomenda-se a utilização do modelo *DRA* em relação ao *SD*, pois este segundo utiliza estruturas (*handles*) e tipos enumerados que resultam numa abstração ao acesso de registradores, o que por um lado torna a programação mais simples, devido ao alto nível de abstração, por outro lado faz com que a mesma não tenha uma eficiência como a do modelo *DRA*.

Num mesmo programa podem ser utilizados ambos os modelos, porém a eficiência ainda assim estará abaixo da programação puramente realizada via *DRA*. Como dito anteriormente, o foco neste tutorial será feito com foco ao modelo de acesso direto ao registro (*DRA*), para consulta ao usuário, está disponível em [4] a documentação que contém todos os registros dos dispositivos TMS320F2837xS.

2.3 Importando e Criando um Projeto

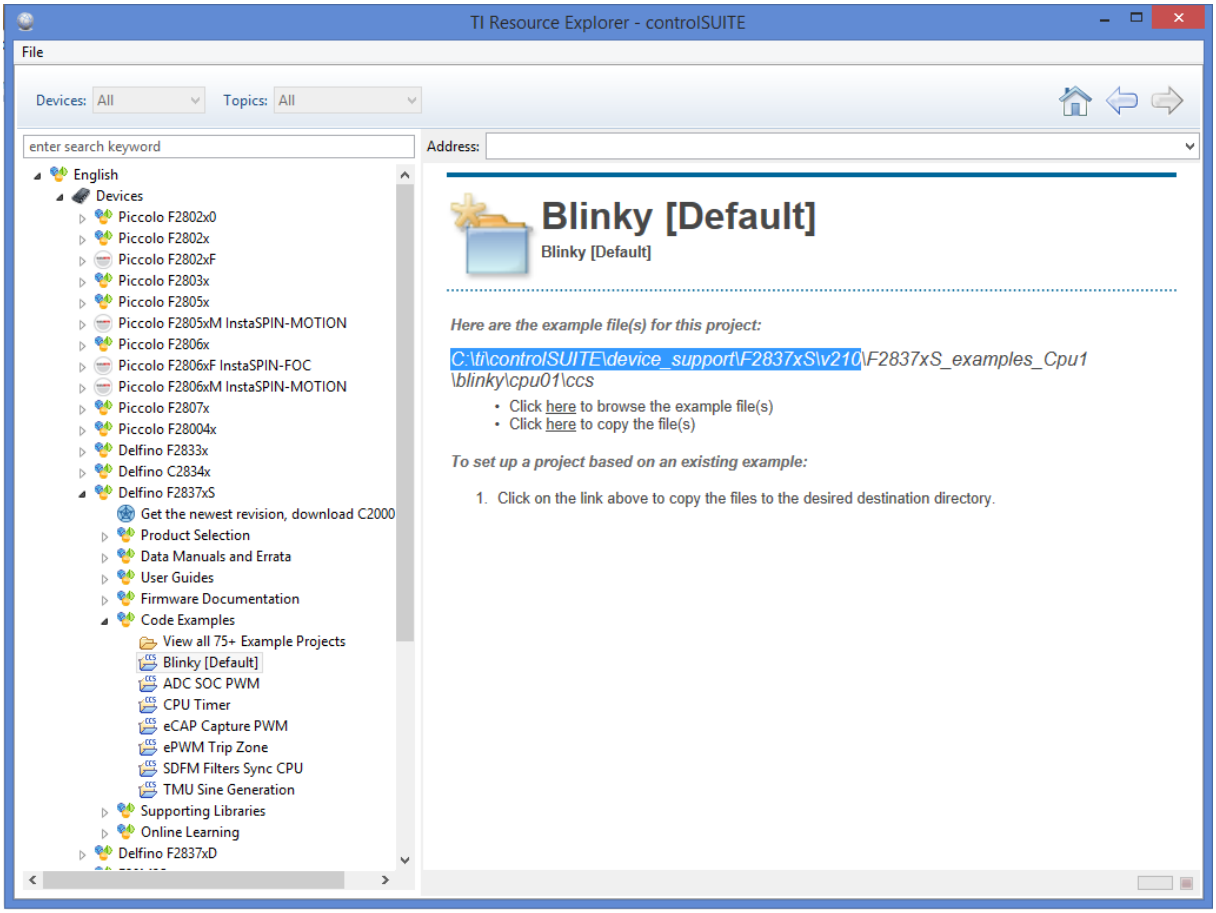
Para a criação de um novo projeto, que será transmitido para a *Launchpad*™, devem ser feitas diversas configurações, que podem vir a demandar uma considerável quantidade de tempo. Para que a criação seja feita mais rapidamente, teremos o auxílio do *ControlSUITE*™*.* Iremos realizar a importação de um projeto exemplo já contido no *ControlSUITE*™ para que todos os *links* de bibliotecas já estejam corretamente anexados ao projeto.

Para importar um projeto exemplo, o usuário deve seguir os seguintes passos descritos abaixo.

1) Executar o software *Code Composer Studio*™ 6.2.0.00050. Caso seja a primeira vez que o usuário utilize a *IDE*, ela irá solicitar que o usuário selecione ou crie um diretório que irá conter o *workspace* do projeto;

2) Executar o software *ControlSUITE*™. Em seguida abrir em sequência, *English ->Devices -> Delfino F2837xS -> Code Examples -> Blinky [Default].* E copiar o diretório que o programa está contido até o diretório *v210*, como mostrado abaixo na Figura 2 .2;

Figura 2.2:Diretório para projetos exemplos do TMS320F28377S.



Fonte: Autoria Própria.

3) De volta ao *Code Composer Studio*™, abra em sequência, *Project -> Import CCS Projects...* e no campo para escrita do diretório, cole o endereço copiado do *ControlSUITE*™ e pressione a tecla *enter*;

4) Após carregar todos os arquivos exemplo, selecione o projeto exemplo que usará como *template* com um *check* dentro de *Discovered projects*. Pode-se escolher mais de um projeto, porém nós iremos trabalhar apenas com um;

5) Selecione o *check* em *Copy projects into workspace* para que uma cópia do projeto seja criada na sua pasta de workspace criada ao iniciar o *Code Composer Studio* pela primeira vez. Após isto clique em *Finish*;

6) Na aba *Project Explorer*, exclua o projeto importado clicando com o botão direito do *mouse* sobre ele e selecionando a opção *Delete*, ou simplesmente apertando o botão *Del* do teclado. Após isto, tenha a certeza de que o *check* de *Delete Project contents on disk* esteja desmarcado, e clique em ok.

7) Repita os passos 3 até 5, porém agora utilizando como diretório o workspace do usuário. Isto é feito para que os projetos originais do *ControlSUITE*™

não sejam alterados, e possamos ter uma cópia já com todos os *links* corretamente anexados ao projeto.

3. Inicialização do *DSP* e *GPIOs*

3.1 Inicialização

Para a inicialização do *DSP*, os programas da *Texas Instruments*™ sempre seguem um mesmo passo-a-passo, que será mostrado neste capítulo. Nos exemplos seguintes deste documento, a inicialização será suprimida de maneira a compactar o tamanho dos exemplos, porém a mesma deve estar presente no código para o funcionamento.

O primeiro passo consiste em incluir a biblioteca do *DSP* via comando *include:*

```
#include "F28x_Project.h" // Inclui biblioteca do DSP
```

Após isto, caso o usuário deseje, antes da *main* pode declarar protótipos de futuras funções, defines e variáveis globais. Começando a *main*, duas funções serão chamadas, a primeira para inicializar o sistema do *DSP* e a segunda para a inicialização do sistema de *GPIO*.

```
void main(void)
{
InitSysCtrl();      // Inicializa o sistema geral
InitGpio();         // Inicialização dos GPIOs
```

Estas funções foram escritas pela própria *Texas Instruments*™, e devido ao escopo deste tutorial, tais códigos não serão apresentados aqui. Caso seja de interesse do programador, toda função, variável ou definição pode ser acessada em seu local de declaração para consulta. Para isto deve-se manter pressionado o botão "*Ctrl*" do teclado e com o botão esquerdo do teclado clicar sobre a função, variável ou definição para que o local de definição da mesma seja aberto.

Continuando a função *main*, as interrupções globais serão desabilitadas, e a tabela *PIE* será inicializada via função. Após isto, as interrupções de *CPU* são desabilitadas e os *flags* limpados.

```
DINT;               // Desabilitar Interrupções globais
InitPieCtrl();      // Inicializar a PIE
IER = 0x0000;       // Desabilita interrupções da CPU
IFR = 0x0000;       // Limpa flags de interrupção
```

Por fim, a tabela *PIE* deve receber ponteiros para no futuro poder acessar interrupções relacionadas aos módulos, como por exemplo, o *ADC* e *ePWM*. Com isto, basta habilitar as interrupções globais e de tempo real.

```
InitPieVectTable(); // Inicializa a tabela da PIE com ponteiros
EINT;               // Habilita interrupções globais
ERTM;                  // Habilita interrupções de tempo real
```

Com isto feito, terminamos o passo-a-passo da inicialização do dispositivo. O código desta inicialização está contido no Apêndice A para caso o usuário deseje copiá-lo e colar em sua *IDE*.

3.2 Definição de *GPIOs*

Para definir a multiplexação de um pino *GPIO* e o configurar, é primeiro necessário que a proteção de escrita nos registros seja retirada via operação *EALLOW*. Todas as instruções para configuração do *GPIO* são feitas nos registradores *GpioCtrlRegs*. Para evitar problemas de *glitch*, a documentação *SPRUHX5E*, disponível em [4], recomenda que os *bits* de *GPIOx*, presentes no registrador *GPyMUXz*, recebam o valor 0x00 primeiramente. No caso, “x” representa o número do *GPIO* utilizado, “y” o módulo em que o *GPIO* está presente, e “z” o número do módulo. Após isto o programador pode definir o *GPIO* como quiser, lembrando sempre de ao final das configurações, escrever o comando *EDIS* para não permitir mais escritas nos registros.

Para forçar valores na saída de um *GPIO*, o registrador utilizado é o *GpioDataRegs*, e o dispositivo contém registradores que já efetuam operações de setar, limpar e *toggle* no *bit* de um *GPIO* especifico.

Abaixo está apresentado um exemplo de código, dentro da *main*, em que os *GPIOs* são definidos como saídas.

```
EALLOW;
GpioCtrlRegs.GPAMUX1.bit.GPIO13 = 0x00;  // Evitar glitch (LED AZUL).
GpioCtrlRegs.GPAGMUX1.bit.GPIO13 = 0x00; // Mux para EPWM7B estar no led azul.
GpioCtrlRegs.GPAMUX1.bit.GPIO13 = 0x01;  // Mux para EPWM7B estar no led azul.
GpioCtrlRegs.GPAMUX1.bit.GPIO12 = 0x00;  // Evitar glitch (LED VERMELHO).
GpioCtrlRegs.GPAGMUX1.bit.GPIO12 = 0x00; // Mux para EPWM7A estar no led
                                         vermelho.
GpioCtrlRegs.GPAMUX1.bit.GPIO12 = 0x01;  // Mux para EPWM7A estar no led
                                         vermelho.
GpioCtrlRegs.GPAPUD.bit.GPIO13 = 1;      // Desabilita pullup.
GpioCtrlRegs.GPAPUD.bit.GPIO12 = 1;      // Desabilita pullup.
GpioCtrlRegs.GPADIR.bit.GPIO13 = 1;      // Define como saída.
GpioCtrlRegs.GPADIR.bit.GPIO12 = 1;      // Define como saída.
GpioCtrlRegs.GPAMUX1.bit.GPIO2 = 0x00;   // Evitar glitch.
GpioCtrlRegs.GPAGMUX1.bit.GPIO2 = 0x00;  // Mux para GPIO2.
GpioCtrlRegs.GPAMUX1.bit.GPIO2 = 0x00;   // Mux para GPIO2.
GpioCtrlRegs.GPAPUD.bit.GPIO2 = 1;       // Desabilita pullup em GPIO2.
GpioCtrlRegs.GPADIR.bit.GPIO2 = 1;       // Define como saída GPIO2.
```

```
EDIS;
GpioDataRegs.GPACLEAR.bit.GPIO2 = 1;     // Limpar valor mostrado em GPIO2.
```

A multiplexação dos pinos pode ser consultada via documentação SPRUI25C disponível em [1]. Para saber em qual bloco de *GPIO* cada um se encontra é necessário consultar a documentação SPRUHX5E, disponível em [4]. A escrita nos *bits* de *GPIOx* nos registradores *GPyPUD* e *GPyDIR*, caso fossem invertidas, iriam causar o resultado inverso ao comentado ao lado da instrução.

4. Conversor Analógico-Digital (*ADC*)

4.1 Introdução

O conversor analógico-digital, ou apenas *ADC* é utilizado para converter sinais analógicos em sinais digitais que possam ser interpretados por máquinas. Para que este conversor consiga criar a melhor ponte possível entre o mundo analógico e digital, é necessário que o número de *bits* da representação digital seja o maior possível.

No *DSP* TMS320F28377S, o *ADC* pode trabalhar tanto em 12 *bits*, quanto em 16. Ele possui dois módulos *ADC*, sendo eles o *ADCA* e o *ADCB*, cada um destes módulos é composto por 6 canais, outros dois canais, *ADCIN14* e *ADCIN15*, não são atribuídos a nenhum dos dois módulos.

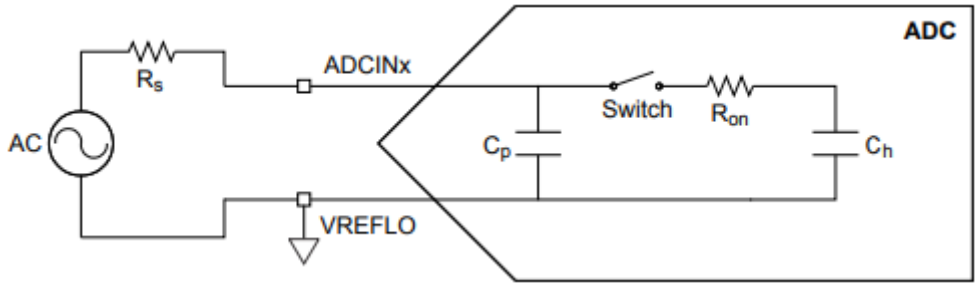
4.2 Princípio de Amostragem e Conversão

Para que um sinal analógico possa ser convertido para digital, o *ADC* usufrui um circuito chamado *Sample & Hold*, ou apenas *S&H*, que realiza a amostragem do sinal. O *S&H* consiste em um circuito RC, e tem a função de carregar um valor de tensão em um capacitor, e esta tensão é a que será entregue para a conversão *AD*. O TMS320F28377S possui um único circuito de *S&H*.

O tempo de amostragem, modo de operação do *ADC*, interrupções relacionadas, dentre outras informações de funcionamento são definidas pelo usuário via escrita em registradores específicos.

A Figura 4 .3apresenta o circuito de *S&H* do dispositivo quando se trabalha com 12 *bits* (*Single-Ended*), e os valores típicos de suas grandezas são: R_{on}=425 Ω (Resistência da chave de amostragem), C_h=14.5 pF (Capacitor de amostragem), R_s=50 Ω (Impedância nominal da fonte) e C_p (Capacitância Parasita) depende do canal escolhido, estes valores de C_p são mostrados na Tabela 5-48 do *Datasheet* do dispositivo. [5]

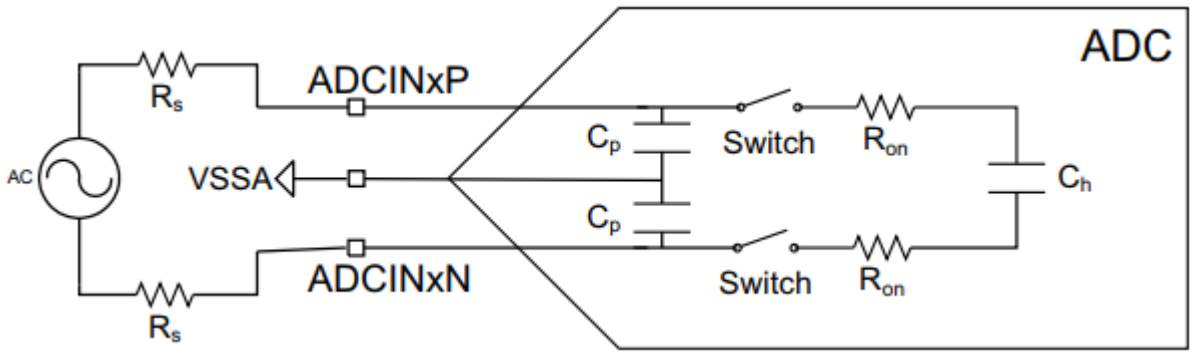
Figura 4.3: Circuito de *S&H* em 12 *bits*.



Fonte: SPRUHX5E. [4]

Quando se trabalha com 16 *bits* (*Differential*), o circuito de *S&H* se torna um pouco diferente, pois neste modo de conversão são necessários dois canais para que a mesma ocorra. O princípio de funcionamento é o mesmo, vale ressaltar apenas que a referência de terra se encontra agora entre as duas capacitâncias parasitas. A Figura 4 .4ilustra o circuito *S&H* para trabalhar em 16 *bits*, e os valores típicos de suas grandezas são: R_{on}=700 Ω (Resistência da chave de amostragem), C_h=16.5 pF (Capacitor de amostragem), R_s=50 Ω (Impedância nominal da fonte) e C_p (Capacitância Parasita) depende do canal escolhido, estes valores de C_p são mostrados na Tabela 5-48 do *Datasheet* do dispositivo. [5]

Figura 4.4: Circuito de *S&H* em 16 *bits*.



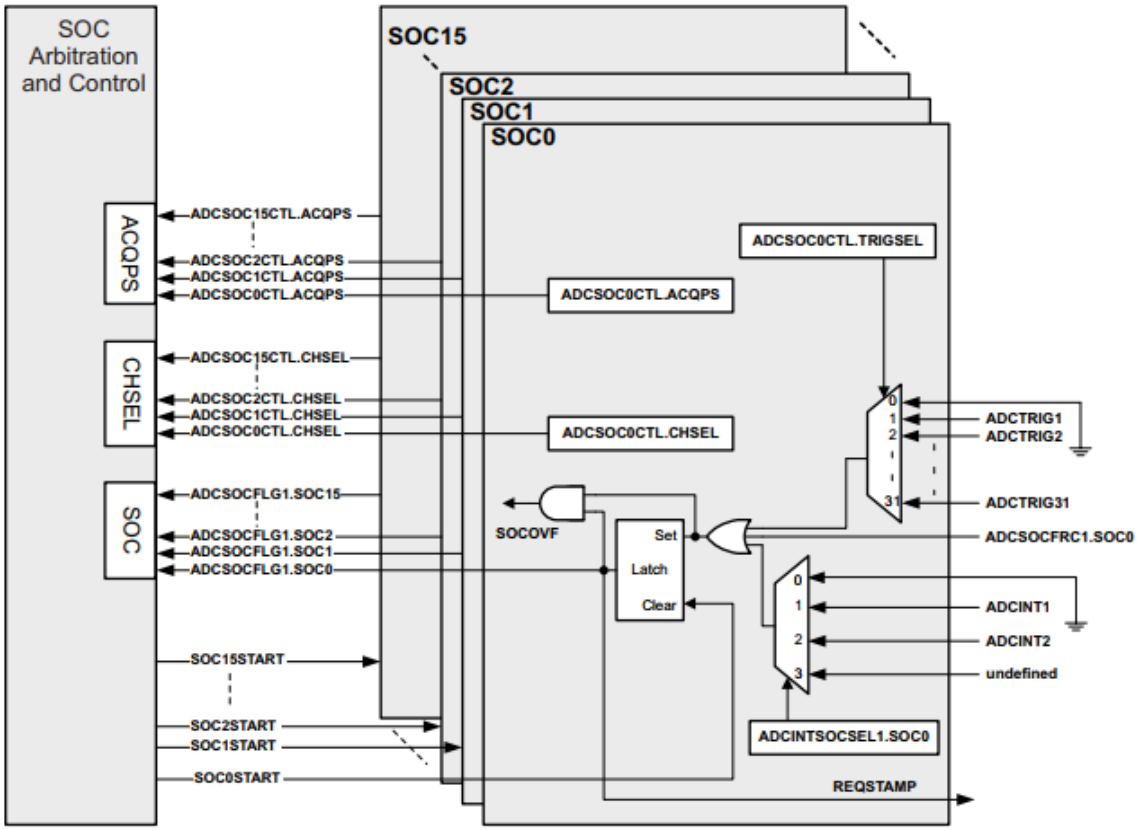
Fonte: SPRUHX5E. [4]

4.3 Start-Of-Conversion

O início da conversão é comandado pelo *SOC* (*Start of Conversion*), que consiste de um pulso, dado por um *trigger* pré-selecionado. O *SOC* pode ter várias fontes, no TMS320F28377S, temos 16 tipos de *SOCs* e 29 diferentes fontes.

A Figura 4 .5 apresenta um diagrama de blocos gerais de operação dos *SOCs*.

Figura 4.5: Diagrama de blocos para os *SOCs*.



Fonte: SPRUHX5E. [4]

Um mesmo canal do *ADC* pode estar associado a mais de um *SOC*, ou um mesmo *SOC* ser responsável pela conversão de vários canais. Caso mais de um canal receba o *SOC* simultaneamente, a prioridade irá depender do critério de *Round Robin*, o qual está detalhado na documentação *SPRUHX5E*.

A seleção de qual *SOC* será associado a cada canal é feita pela escrita no conjunto de registradores *AdcyRegs*, onde “y” é o módulo do *ADC*, sendo o A ou B. Continuando o processo para selecionar o *SOC*, a escrita deve ser realizada no registrador *ADCSOCxCTL*, onde x é o número do *SOC* o qual o usuário deseja trabalhar, setando o *bit* de *CHSEL* em nível lógico alto. Por exemplo, caso o usuário queira utilizar o *SOC7* para converter o canal *ADCINA3*, basta realizar a instrução: *AdcaRegs.ADCSOC7CTL.bit.CHSEL* = 1;

4.4 Inicialização do *ADC*

Para a inicialização do módulo *ADC*, primeiramente deve-se utilizar a instrução *EALLOW* para permitir a escrita nos registradores, e após isto, escolher o *prescale* de operação do módulo com relação ao valor nominal de *ADCCLK*, que é de 100 MHz. Esta seleção é feita pela escrita nos *bits* de *PRESCALE*, presente no registador *ADCCTL2* do módulo trabalhado do *ADC*.

Os programas exemplos da *Texas Instruments*™ fazem uso de uma função própria para setar as características de operação e conversão do módulo *ADC*, chamada *AdcSetMode*. A função *AdcSetMode* possui três parâmetros, o módulo a ser configurado (A ou B), a resolução (12 ou 16 *bits*), e modo de operação (*Single-Ended ou Differential*).

Agora, basta inicializar o *ADC* via escrita de nível lógico alto ao *bitADCPWDNZ* presente no registrador *ADCCTL1* do módulo trabalhado do *ADC*, e para garantir a correta inicialização do módulo, é necessário um *delay* de 1 ms, feito pela função *DELAY_US*, que possui um parâmetro apenas, que é o tempo de *delay* em µs. Por fim, basta o comando *EDIS* para impedir a escrita nos registradores protegidos.

Para selecionar informações do pulso de *trigger* de interrupção, a mesma deve ser feita antes de inicializar o módulo, porém será trabalhada mais adiante.

4.5 Caracterização do *ADC*

Após a inicialização correta do *ADC*, devemos definir o valor mínimo ciclos de *SYSCLK* necessários para que a janela de amostragem para a resolução escolhida seja suficiente. A janela de amostragem é o tempo em que o circuito de *S&H* irá amostrar o sinal. Segundo programas exemplos da *Texas Instruments*™, para 12 *bits*, são necessários 15 ciclos de *SYSCLK* e para 16 *bits*, 63 ciclos.

Agora deve-se retirar a proteção de escrita em registradores com a instrução *EALLOW*, e escolher quais canais e *SOCs* serão utilizados. Na prática, a instrução terá a seguinte sintaxe: *AdcyRegs.ADCSOCxCTL.bit.CHSEL* = z;. Onde “y” é o

módulo *ADC* A ou B, “x” é o número do *SOC* utilizado e “z” é o conjunto de *bits* que definirá o canal a ser convertido.

4.6 Interrupções do *ADC*

Para que a interrupção do *ADC* seja realizada, é primordial que o programador defina a posição do pulso de interrupção na escrita do *bit* de *INTPULSEPOS* do registrador *ADCCTL1*. Caso a escrita seja “0”, o pulso será gerado ao final da janela de aquisição, caso a escrita seja “1”, o pulso ocorrerá ao final da conversão (*EOC*) com um 1 ciclo de atraso para os resultados do *ADC* serem armazenados no registro de resultados.

Após o pulso ser gerado, o *flag* da interrupção será setado em nível lógico alto, e deve ser limpo para que a *CPU* entenda que a interrupção já foi tratada. Este processo é realizado via escrita no *bitADCINTn*, onde “n” é o número de 1 a 4 da interrupção selecionada, que é definida pelos registradores de *ADCINTSEL1N2* e *ADCINTSEL3N4*. *ADCINTn* preferencialmente é limpo via escrita de 1 no *bit* de *ADCINTn* no registro *ADCINTFLGCLR*.

Abaixo temos um exemplo de seleção de interrupção e limpeza de *flag*.

```
AdcaRegs.ADCCTL1.bit.INTPULSEPOS = 1; // Pulso de interrupção ao final da janela
                                      de amostragem (final de SOC)
... // Código de programa
AdcaRegs.ADCINTSEL1N2.bit.INT1SEL = 1;  // Final de SOC1 setará flag INT1
AdcaRegs.ADCINTSEL1N2.bit.INT1E = 1;    // Habilita flag INT1
AdcaRegs.ADCINTFLGCLR.bit.ADCINT1 = 1; // Limpa flag de INT1
...// Código de programa
...// Acontecimento da interrupção e tratamento da mesma
AdcaRegs.ADCINTFLGCLR.bit.ADCINT1 = 1; // Limpa flag de INT1
```

Para o tratamento de interrupções também podemos utilizar funções, para isto, elas devem ser definidas nos protótipos de funções, usualmente definidas como:

```
__interrupt void NOME_DA_FUNÇÃO_isr(void);
```

E após a inicialização do *DSP* como um todo, ela deve ser relacionada à tabela *PIE*. Abaixo está um exemplo da linkagem de interrupções do *ADCA* (*INT1*) e de *ePWM7*.

```
EALLOW;
```

```
PieVectTable.ADCA1_INT = &adca1_isr; // Função para interrupção do ADCA linkada
                                        com a tabela PIE.
PieVectTable.EPWM7_INT = &epwm7_isr; // Função para interrupção do ePWM7 linkada
                                        com a tabela PIE.
EDIS;
```

Também é necessário que certos grupos de interrupções sejam habilitados como mostrado abaixo. O grupo em que cada interrupção está contida deve ser consultado via "*Ctrl*+Botão esquerdo" na tabela *PIE*.

```
IER |= M_INT1; // Habilita o grupo 1 de interrupções, serve para o ADC
IER |= M_INT3; // Habilita o grupo 3 de interrupções, serve para o ePWM
EINT;          // Habilita as interrupções globais.
ERTM;          // Habilita as interrupções de tempo real.
```

Por fim a interrupção em si é habilitada para que ocorra.

```
PieCtrlRegs.PIEIER1.bit.INTx1 = 1; // Habilita interrupção do ADCA
PieCtrlRegs.PIEIER3.bit.INTx7 = 1; // Habilita interrupção do ePWM7
```

As funções de interrupção contêm o código de tratamento do processo, e ao final de sua execução, devem limpar o *flag* da interrupção em questão e permitir que outras interrupções sejam notadas via escrita em registrador como mostrado abaixo.

```
AdcaRegs.ADCINTFLGCLR.bit.ADCINT1 = 1;  // Limpa o flag de interrupção
PieCtrlRegs.PIEACK.all = PIEACK_GROUP1; // Permite que outras interrupções da
natureza do Grupo 1 ocorram novamente
```

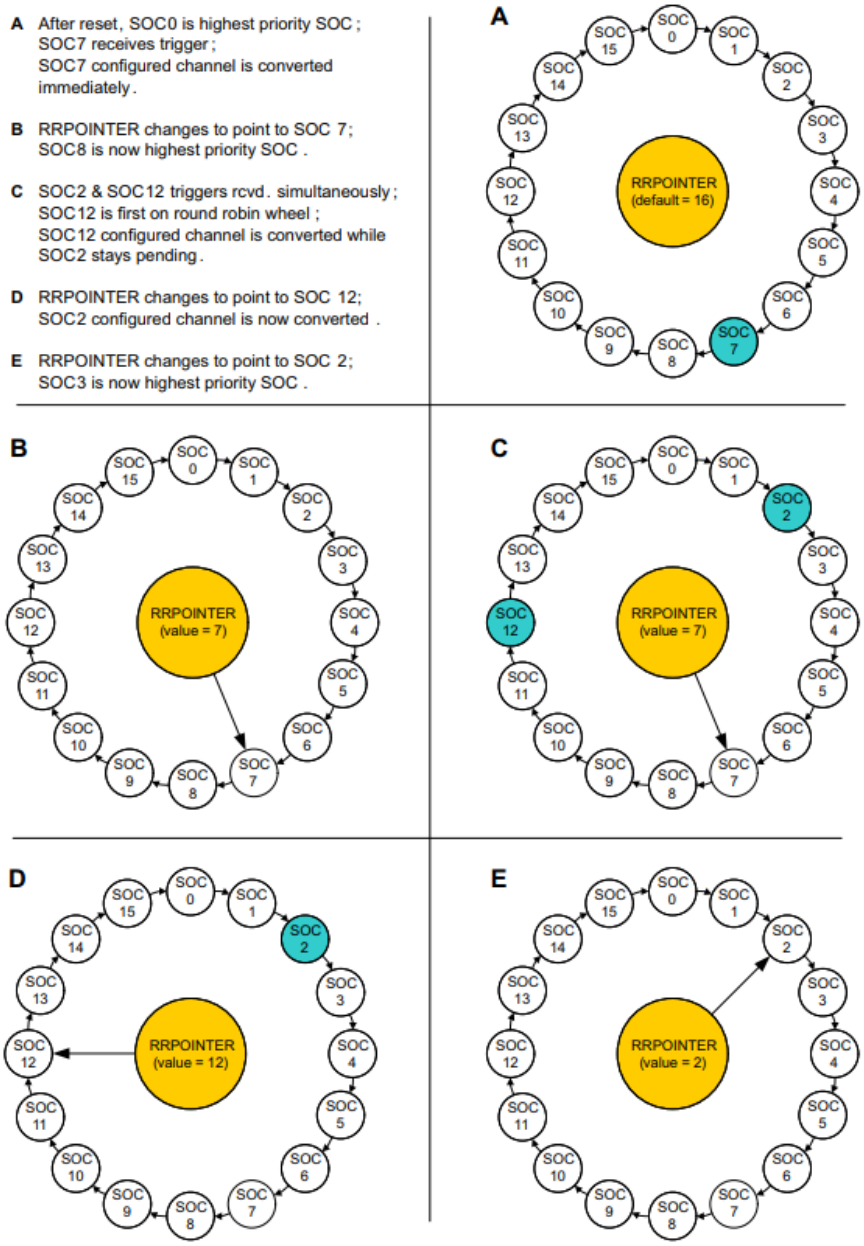
4.7 Prioridade de Conversão (*Round Robin*)

O *ADC* realiza apenas uma conversão de cada vez, e a prioridade de cada uma segue a *Round Robin*, ou ciclo de *Robin*, que é um critério de prioridade. Ela entra em prática quando dois, ou mais *flags de SOCs* são setados ao mesmo tempo, e é dada a prioridade de conversão de acordo com o conteúdo do ponteiro *RRPOINTER* (*Round Robin Pointer*). A prioridade maior de conversão é dada ao *SOC* apontado por RRPOINTER, a segunda maior prioridade para o próximo *SOC* e assim por diante. Na Figura 4 .6 está mostrado um exemplo da aplicação de *Round Robin*, dividida nos momentos A até E, que ocorrem em sequência.

No exemplo da Figura 4 .6 no momento A, após o *reset*, por *default*, *RRPOINTER* aponta para o endereço de *SOC0*, sendo assim ele é o *SOC* com maior prioridade. Após isto, um *trigger* aciona *SOC7*, e como não há outro *flag* de *SOC* ativo, *SOC7* imediatamente começa a ser convertido. No momento B, após a conversão, *RRPOINTER* agora aponta para o endereço de *SOC7*, fazendo com que a maior prioridade seja de *SOC8*. No momento C, *SOC2* e *SOC12* recebem seus

respectivos *triggers* simultaneamente, porém devido à prioridade do *RRPOINTER*, o *SOC12* será convertido primeiro, e o *SOC2* ficará com pendência na conversão.

Figura 4.6: Exemplo de *Round Robin*.



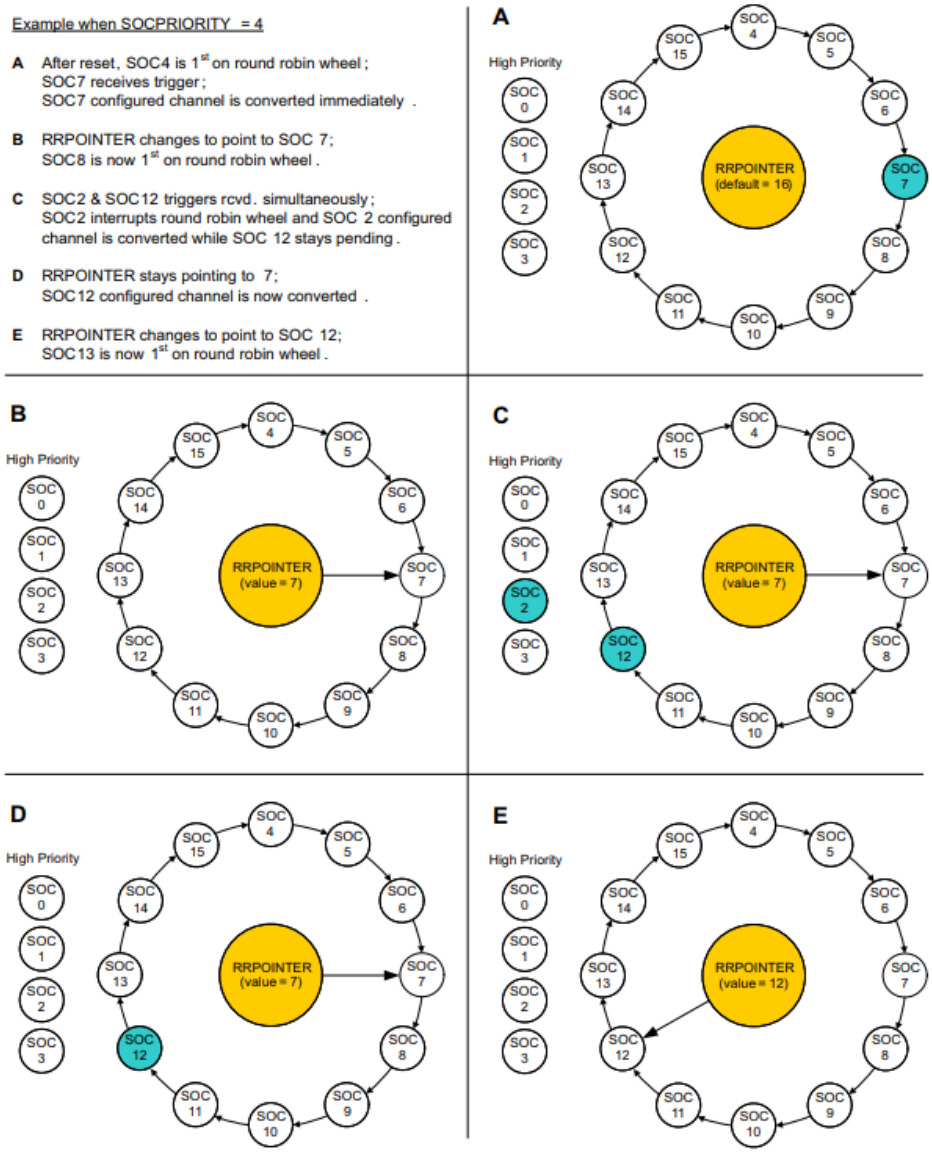
Fonte: SPRUHX5E. [4]

No momento D, o *RRPOINTER* aponta agora para o endereço *SOC12,* e como não existem conversões com prioridade acima de *SOC2*, ela será convertida agora. No momento E, após a conversão de *SOC2*, o *RRPOINTER* aponta para o endereço de *SOC2*, dando prioridade maior para uma conversão de *SOC3*.

O usuário pode também selecionar *SOCs* que terão alta prioridade independente do endereço apontado por *RRPOINTER*. Para isto, campo *SOCPRIORITY* no registrador *ADCSOCPRIORITYCTL* deve ser preenchido de acordo com a necessidade do usuário. O valor escrito em *SOCPRIORITY* representa a partir de qual *SOC* a prioridade passará a depender do *Round Robin*. Caso seja escrito “2” no campo de *SOCPRIORITY*, os *SOCs* “0” e “1” terão alta prioridade fixa, enquanto os outros dependerão do *Round Robin*. Esta alta prioridade fixa significa que independente do endereço apontado por *RRPOINTER*, se um dos *SOCs* prioritários receber seu *trigger*, a próxima conversão será desta alta prioridade. Caso mais de um *SOC* prioritário receba o *trigger* simultaneamente, a prioridade será dada em ordem decrescente a partir de *SOC0*.

Figura 4 .7 exemplifica o caso de alta prioridade fixa e é dividido em momentos A até E, assim como na Figura 4 .6 Neste exemplo, o valor de *SOCPRIORITY* é “4”, sendo assim os *SOCs* “0” a “3” possuem alta prioridade com relação aos demais, que serão priorizados de acordo com o *Round Robin.* No momento A, o *SOC* com maior prioridade devido à *RRPOINTER* é o *SOC4*. O *SOC7* receber seu *trigger* e é convertido. No momento B, *RRPOINTER* agora aponta para o endereço de *SOC7*, dando a prioridade para *SOC8*. No momento C, ambos os *SOCs* “2” e “12” recebem seus *triggers* simultaneamente. Devido à alta prioridade de *SOC2* devido aos *bits* de *SOCPRIORITY*, a primeira conversão a ser feita será de *SOC2*, e *SOC12* ficará como pendente à conversão. No momento D, *RRPOINTER* continua apontando ao endereço de *SOC7*, e *SOC12* será convertido. No momento E, após a conversão de *SOC12*, agora *RRPOINTER* está priorizando a conversão de *SOC13*.

Figura 4.7: Exemplo de *Round Robin* em alta prioridade.



Fonte: SPRUHX5E. [4]

4.8 Exemplo 1 – Conversão Contínua

Este exemplo irá tratar da conversão *ADC* contínua, sem o auxílio do módulo *ePWM*, ou outros módulos auxiliares. As inicializações da seção 3.1 já foram feitas

para suprimir o código, e abaixo estão listadas apenas as funções protótipo, variáveis globais e *defines*.

```
void ConfigureADC(void);                    // Função que irá inicializar o ADC
void SetupADCContinuous(Uint16 channel); // Função que irá configurar o ADC
#define RESULTS_BUFFER_SIZE 256 // Tamanho do Vetor de resultados
Uint16 AdcaResults[RESULTS_BUFFER_SIZE]; // Vetor de resultados
Uint16 resultsIndex;
```

Vamos começar inicializando o *ADC* com a função *ConfigureADC*.

```
ConfigureADC(); // Inicialização do ADC
```

A função *ConfigureADC* irá inicializar o módulo de uma maneira específica, dentro das possibilidades de configuração já apresentadas.

```
void ConfigureADC(void)
{
EALLOW;
AdcaRegs.ADCCTL2.bit.PRESCALE = 6; // Divide ADCCLK por 4
// Configura o ADCA para trabalhar em 12-bits e em modo single-ended
AdcSetMode(ADC_ADCA, ADC_RESOLUTION_12BIT, ADC_SIGNALMODE_SINGLE);
// O pulso de trigger da interrupção será dado após o final da conversão
AdcaRegs.ADCCTL1.bit.INTPULSEPOS = 1;
AdcaRegs.ADCCTL1.bit.ADCPWDNZ = 1; // Energiza o ADC
// Delay de 1 ms para garantir inicialização total do ADC
DELAY_US(1000);
EDIS;
}
```

Na função *ConfigureADC*, primeiramente utilizamos o comando *EALLOW* para permitir a escrita nos registradores protegidos. A escrita nos *bits* de *PRESCALE* configuram a frequência do *clock* do *ADC*, no caso estamos dividindo *ADCCLK* por 4. A função *AdcSetMode* também é escrita pela *Texas Instruments*™ e tem como finalidade inicializar um dos módulos do *ADC* em uma configuração específica, no caso deste exemplo, o *ADC* A será trabalhado em 12 *bits* e em modo *Single-Ended*. As interrupções do *ADC* serão interpretadas após a conversão devido ao valor do *bit* de *INTPULSEPOS*. Tendo todas estas configurações feitas, ligamos o *ADC* e esperamos 1 ms para garantir que o mesmo tenha tempo hábil para inicializar. Para terminar, realizamos a desabilitação da escrita destes registradores com o comando *EDIS*.

```
SetupADCContinuous(0); // Canal 0 em conversões contínuas
```

Voltando ao código da *main*, a função *SetupADCContinuous* será chamada, para configurar conversões contínuas do canal 0 do módulo de *ADC* A. As instruções posteriores servem apenas para habilitar as interrupções que foram desabilitadas antes no código. Dependendo da resolução do canal do módulo *ADC* em questão, será atribuído um valor mínimo de *acqps* para a janela de amostragem.

```
void SetupADCContinuous(Uint16 channel)
```

```
{
Uint16 acqps;
// Define a janela de aquisição mínima em SYSCLKS de acordo com a resolução
if(ADC_RESOLUTION_12BIT == AdcaRegs.ADCCTL2.bit.RESOLUTION){
acqps = 14; // 75 ns
}
else { // Resolução 16 bits
acqps = 63; // 320 ns
}
```

Todos os *SOCs* irão ser responsáveis pela conversão no mesmo canal *channel*, no caso, o canal 0 do módulo *ADC* A.

```
EALLOW;
AdcaRegs.ADCSOC0CTL.bit.CHSEL  = channel;  //SOC irá converter em channel
AdcaRegs.ADCSOC1CTL.bit.CHSEL  = channel;
AdcaRegs.ADCSOC2CTL.bit.CHSEL  = channel;
AdcaRegs.ADCSOC3CTL.bit.CHSEL  = channel;
AdcaRegs.ADCSOC4CTL.bit.CHSEL  = channel;
AdcaRegs.ADCSOC5CTL.bit.CHSEL  = channel;
AdcaRegs.ADCSOC6CTL.bit.CHSEL  = channel;
AdcaRegs.ADCSOC7CTL.bit.CHSEL  = channel;
AdcaRegs.ADCSOC8CTL.bit.CHSEL  = channel;
AdcaRegs.ADCSOC9CTL.bit.CHSEL  = channel;
AdcaRegs.ADCSOC10CTL.bit.CHSEL = channel;
AdcaRegs.ADCSOC11CTL.bit.CHSEL = channel;
AdcaRegs.ADCSOC12CTL.bit.CHSEL = channel;
AdcaRegs.ADCSOC13CTL.bit.CHSEL = channel;
AdcaRegs.ADCSOC14CTL.bit.CHSEL = channel;
AdcaRegs.ADCSOC15CTL.bit.CHSEL = channel;
```

O valor de *acqps* é atribuído a todos os *SOCs*, fazendo com que a janela seja constituída de 1 + *acqps* ciclos de *SYSCLK*.

```
//Janela de amostragem é acqps + 1 ciclo de SYSCLK
AdcaRegs.ADCSOC0CTL.bit.ACQPS  = acqps;
AdcaRegs.ADCSOC1CTL.bit.ACQPS  = acqps;
AdcaRegs.ADCSOC2CTL.bit.ACQPS  = acqps;
AdcaRegs.ADCSOC3CTL.bit.ACQPS  = acqps;
AdcaRegs.ADCSOC4CTL.bit.ACQPS  = acqps;
AdcaRegs.ADCSOC5CTL.bit.ACQPS  = acqps;
AdcaRegs.ADCSOC6CTL.bit.ACQPS  = acqps;
AdcaRegs.ADCSOC7CTL.bit.ACQPS  = acqps;
AdcaRegs.ADCSOC9CTL.bit.ACQPS  = acqps;
AdcaRegs.ADCSOC10CTL.bit.ACQPS = acqps;
AdcaRegs.ADCSOC11CTL.bit.ACQPS = acqps;
AdcaRegs.ADCSOC12CTL.bit.ACQPS = acqps;
AdcaRegs.ADCSOC13CTL.bit.ACQPS = acqps;
AdcaRegs.ADCSOC14CTL.bit.ACQPS = acqps;
AdcaRegs.ADCSOC15CTL.bit.ACQPS = acqps;
```

Os *flags* das interrupções 1 a 4 são desabilitados para evitar que as conversões comecem antes da configuração terminar, e também garante que outras interrupções causadas pelo mesmo *trigger* só ocorrerão quando o *flag* referente for limpado.

```
AdcaRegs.ADCINTSEL1N2.bit.INT1E = 0; //Desabilita flag de INT1
AdcaRegs.ADCINTSEL1N2.bit.INT2E = 0; //Desabilita flag de INT2
AdcaRegs.ADCINTSEL3N4.bit.INT3E = 0; //Desabilita flag de INT3
```

```
AdcaRegs.ADCINTSEL3N4.bit.INT4E = 0; //Desabilita flag de INT4
//Só serão gerados novos pulsos de ADCINT1 quando o flag for limpado
AdcaRegs.ADCINTSEL1N2.bit.INT1CONT = 0;
//Só serão gerados novos pulsos de ADCINT2 quando o flag for limpado
AdcaRegs.ADCINTSEL1N2.bit.INT2CONT = 0;
//Só serão gerados novos pulsos de ADCINT3 quando o flag for limpado
AdcaRegs.ADCINTSEL3N4.bit.INT3CONT = 0;
//Só serão gerados novos pulsos de ADCINT4 quando o flag for limpado
AdcaRegs.ADCINTSEL3N4.bit.INT4CONT = 0;
```

Aqui são selecionados quais os *SOCs* irão causar o *set* dos *flags* de interrupção. Também serão selecionados os triggers para os 16 *SOCs*.

```
AdcaRegs.ADCINTSEL1N2.bit.INT1SEL = 6;  //O fim de SOC6  irá setar flag INT1
AdcaRegs.ADCINTSEL1N2.bit.INT2SEL = 14; //O fim de SOC14 irá setar flag INT2
AdcaRegs.ADCINTSEL3N4.bit.INT3SEL = 7;  //O fim de SOC7  irá setar flag INT3
AdcaRegs.ADCINTSEL3N4.bit.INT4SEL = 15; //O fim de SOC15 irá setar flag INT4
//ADCINT2 será o trigger para os 8 primeiros SOCs
AdcaRegs.ADCINTSOCSEL1.bit.SOC0 = 2;
AdcaRegs.ADCINTSOCSEL1.bit.SOC1 = 2;
AdcaRegs.ADCINTSOCSEL1.bit.SOC2 = 2;
AdcaRegs.ADCINTSOCSEL1.bit.SOC3 = 2;
AdcaRegs.ADCINTSOCSEL1.bit.SOC4 = 2;
AdcaRegs.ADCINTSOCSEL1.bit.SOC5 = 2;
AdcaRegs.ADCINTSOCSEL1.bit.SOC6 = 2;
AdcaRegs.ADCINTSOCSEL1.bit.SOC7 = 2;
//ADCINT1 será o trigger para os próximos 8 SOCs
AdcaRegs.ADCINTSOCSEL2.bit.SOC8 = 1;
AdcaRegs.ADCINTSOCSEL2.bit.SOC9 = 1;
AdcaRegs.ADCINTSOCSEL2.bit.SOC10 = 1;
AdcaRegs.ADCINTSOCSEL2.bit.SOC11 = 1;
AdcaRegs.ADCINTSOCSEL2.bit.SOC12 = 1;
AdcaRegs.ADCINTSOCSEL2.bit.SOC13 = 1;
AdcaRegs.ADCINTSOCSEL2.bit.SOC14 = 1;
AdcaRegs.ADCINTSOCSEL2.bit.SOC15 = 1;
}
```

Tendo terminado a função, voltemos à *main*. O vetor de resultados será limpo para nos livrarmos de possíveis “lixos” nos valores armazenados nele.

```
// Inicializar o vetor de resultados
for(resultsIndex = 0; resultsIndex < RESULTS_BUFFER_SIZE; resultsIndex++)
{
AdcaResults[resultsIndex] = 0;
}
resultsIndex = 0;
```

Após isto, terminamos todas as configurações necessárias, e entraremos num *loop* infinito para as conversões do *ADC*. Os *flags* de interrupção serão habilitados para iniciar a amostragem, e também eles serão limpos, para garantir que as interrupções e os *SOCs* possam ocorrer.

```
do
{
// Habilitar Flags
AdcaRegs.ADCINTSEL1N2.bit.INT1E = 1;
AdcaRegs.ADCINTSEL1N2.bit.INT2E = 1;
AdcaRegs.ADCINTSEL3N4.bit.INT3E = 1;
```

```
AdcaRegs.ADCINTSEL3N4.bit.INT4E = 1;
AdcaRegs.ADCINTFLGCLR.all = 0x000F; // Limpa flags
```

Agora iremos inicializar o índice do vetor de resultados, e também forçar os *SOCs* 0 a 7.

```
// Inicializa o índice
resultsIndex = 0;
//Software força SOCs 0 à 7
AdcaRegs.ADCSOCFRC1.all = 0x00FF;
```

Será inicializado agora outro loop, para preencher o vetor de resultados até que o mesmo se encontre completamente populado por valores amostrados pelo *ADC*. Assim que as primeiras 8 conversões forem efetuadas, os *flags* com relação às interrupções 1 e 3 do *ADC* serão limpados, e os valores de conversão serão atribuídos ao vetor de resultados. Semelhantemente o mesmo processo é repetido para as próximas 8 conversões, totalizando no total 16 valores.

```
// Continua amostrando até que o vetor de resultados esteja cheio
while(resultsIndex < RESULTS_BUFFER_SIZE)
{
// Espera até que as 8 primeiras conversões sejam completadas
while(0 == AdcaRegs.ADCINTFLG.bit.ADCINT3);
// Limpa ambos os flags de INT pelas primeiras 8 conversões
AdcaRegs.ADCINTFLGCLR.bit.ADCINT1 = 1;
AdcaRegs.ADCINTFLGCLR.bit.ADCINT3 = 1;
// Salva resultados no vetor
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT0;
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT1;
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT2;
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT3;
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT4;
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT5;
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT6;
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT7;
// Espera pelas próximas 8 conversões
while(0 == AdcaRegs.ADCINTFLG.bit.ADCINT4);
// Limpa ambos os flags de INT pelas 8 conversões decorridas
AdcaRegs.ADCINTFLGCLR.bit.ADCINT2 = 1;
AdcaRegs.ADCINTFLGCLR.bit.ADCINT4 = 1;
// Salva resultados no vetor
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT8;
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT9;
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT10;
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT11;
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT12;
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT13;
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT14;
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT15;
}
```

Após 16 repetições do processo, o vetor de resultados irá se encontrar totalmente populado. Sendo assim os *flags* das interrupções serão desabilitados para interromper a amostragem, e o software executa um *breakpoint*, caso o usuário quiser consultar os valores da conversão via *IDE*. Caso seja de interesse uma

conversão contínua sem paradas, basta remover as desabilitações de *flags* e o comando de *breakpoint*. Desta maneira o código iria re-popular o vetor com os novos valores amostrados.

```
// Desabilita flags para interromper a amostragem
AdcaRegs.ADCINTSEL1N2.bit.INT1E = 0;
AdcaRegs.ADCINTSEL1N2.bit.INT2E = 0;
AdcaRegs.ADCINTSEL3N4.bit.INT3E = 0;
AdcaRegs.ADCINTSEL3N4.bit.INT4E = 0;
asm("   ESTOP0"); // breakpoint de software
}while(1);
}
```

O código completo do Exemplo 1 se encontra no Apêndice B ao final deste documento para a cópia do mesmo.

4.9 Exemplo 2 – Conversão com *SOC* via *software*

O próximo exemplo é uma conversão onde o *SOC* será dado via *software*. Assim como no primeiro exemplo, a inicialização do *DSP* já foi feita, e vamos apresentar apenas as definições de funções, variáveis globais e *defines*.

```
void ConfigureADC(void);      // Função que irá inicializar o ADC
void SetupADCSoftware(void);// Função que irá configurar o ADC
// Variáveis que armazenarão os resultados de conversão
Uint16 AdcaResult0;
Uint16 AdcaResult1;
Uint16 AdcbResult0;
Uint16 AdcbResult1;
```

Vamos agora partir para a função *main*, após a declaração das inicializações já comentadas na seção 3.1.

```
ConfigureADC();           // Inicialização do ADC
```

A função *ConfigureADC* irá inicializar os *ADCs* utilizados no exemplo, serão inicializados os módulos A e B, ambos serão trabalhados em 12 *bits* e em modo *Single-Ended*. Os *clocks* serão divididos por “4”, e os pulsos de *trigger* serão dados apenas após o término da conversão. O *delay* de 1 ms existe para permitir que o *ADC* possua tempo hábil para inicializar.

```
void ConfigureADC(void){
EALLOW;
AdcaRegs.ADCCTL2.bit.PRESCALE = 6; // Divide ADCCLK por 4
AdcbRegs.ADCCTL2.bit.PRESCALE = 6; // Divide ADCCLK por 4
// Configura o ADCA e ADCB para trabalhar em 12 bits e em single-ended
AdcSetMode(ADC_ADCA, ADC_RESOLUTION_12BIT, ADC_SIGNALMODE_SINGLE);
AdcSetMode(ADC_ADCB, ADC_RESOLUTION_12BIT, ADC_SIGNALMODE_SINGLE);
// O pulso de trigger da interrupção sera dado após o final da conversão
AdcaRegs.ADCCTL1.bit.INTPULSEPOS = 1;
```

```
AdcbRegs.ADCCTL1.bit.INTPULSEPOS = 1;
AdcaRegs.ADCCTL1.bit.ADCPWDNZ = 1; // Energiza o ADC
AdcbRegs.ADCCTL1.bit.ADCPWDNZ = 1; // Energiza o ADC
DELAY_US(1000);// Delay de 1 ms para garantir inicialização total do ADC
EDIS;
}
```

Voltando à *main*, a próxima instrução é a chamada da função *SetupADCSoftware*.

```
SetupADCSoftware(); // Configurar o ADC
```

A função *SetupADCSoftware* começa definindo o valor da variável *acqps* que será o número mínimo de janelas + 1 de *SYSCLK* para que a amostragem numa certa resolução ocorra de maneira correta. Após isto, a instrução *EALLOW* irá permitir a escrita em registradores protegidos. Para o *ADCA* e *ADCB*, serão escolhidos *SOC0* para converter os pinos A0 e B0, e o *SOC1* para os pinos A1 e B1. A janela de amostragem será de *acqps* + 1 ciclos de *SYSCLK* para todos os pinos, e o final de *SOC0* e *SOC1* irão gerar pulsos de interrupção com relação ao *flag* de *INT1*.

```
void SetupADCSoftware(void)
{
Uint16 acqps;
// Determina janela minima para amostragem em SYSCLK com base na resolução
if(ADC_RESOLUTION_12BIT == AdcaRegs.ADCCTL2.bit.RESOLUTION){
acqps = 14; // 75 ns
}
else { // Resolução é 16 bits
acqps = 63; // 320 ns
}
// Seleciona os canais para conversão e flag de EOC para o ADCA
EALLOW;
AdcaRegs.ADCSOC0CTL.bit.CHSEL = 0;  // SOC0 irá converter pino A0
AdcaRegs.ADCSOC0CTL.bit.ACQPS = acqps; // Janela de amostragem é acqps+1
   ciclos de SYSCLK
AdcaRegs.ADCSOC1CTL.bit.CHSEL = 1;     // SOC1 irá converter pino A1
AdcaRegs.ADCSOC1CTL.bit.ACQPS = acqps; // Janela de amostragem é acqps+1
    ciclos de SYSCLK
AdcaRegs.ADCINTSEL1N2.bit.INT1SEL = 1; // Fim de SOC1 irá setar o flag INT1
AdcaRegs.ADCINTSEL1N2.bit.INT1E = 1;   // Habilita flag de INT1
AdcaRegs.ADCINTFLGCLR.bit.ADCINT1 = 1; // Limpa flag de INT1
// Seleciona os canais para conversão e flag de EOC para o ADCB
AdcbRegs.ADCSOC0CTL.bit.CHSEL = 0;  // SOC0 irá converter pino B0
AdcbRegs.ADCSOC0CTL.bit.ACQPS = acqps; // Janela de amostragem é acqps+1
    ciclos de SYSCLK
AdcbRegs.ADCSOC1CTL.bit.CHSEL = 1;     // SOC1 irá converter pino B1
AdcbRegs.ADCSOC1CTL.bit.ACQPS = acqps; // Janela de amostragem é acqps+1
    ciclos de SYSCLK
AdcbRegs.ADCINTSEL1N2.bit.INT1SEL = 1; // Fim de SOC1 irá setar o flag INT1
AdcbRegs.ADCINTSEL1N2.bit.INT1E = 1;   // Habilita flag de INT1
AdcbRegs.ADCINTFLGCLR.bit.ADCINT1 = 1; // Limpa flag de INT1
}
```

De volta a *main*, iremos inicializar o *loop* infinito para as conversões do *ADC*. Primeiramente serão forçados os *SOCs* para ambos os módulos *ADC*. Será esperado o término de ambas as conversões, após isto, os *flags* serão limpos e os resultados armazenados nas variáveis globais declaradas antes da *main*.

```
do
{
// Força SOC0 e SOC1 para ADCA e ADCB, inicializando as conversões
AdcaRegs.ADCSOCFRC1.all = 0x0003;
AdcbRegs.ADCSOCFRC1.all = 0x0003;
// Espera término da conversão de ADCA, e logo após limpa o flag
while(AdcaRegs.ADCINTFLG.bit.ADCINT1 == 0);
AdcaRegs.ADCINTFLGCLR.bit.ADCINT1 = 1;
// Espera término da conversão de ADCB, e logo após limpa o flag
while(AdcbRegs.ADCINTFLG.bit.ADCINT1 == 0);
AdcbRegs.ADCINTFLGCLR.bit.ADCINT1 = 1;
// Armazena resultados
AdcaResult0 = AdcaResultRegs.ADCRESULT0;
AdcaResult1 = AdcaResultRegs.ADCRESULT1;
AdcbResult0 = AdcbResultRegs.ADCRESULT0;
AdcbResult1 = AdcbResultRegs.ADCRESULT1;
asm("   ESTOP0"); // Breakpoint
}while(1);
}
```

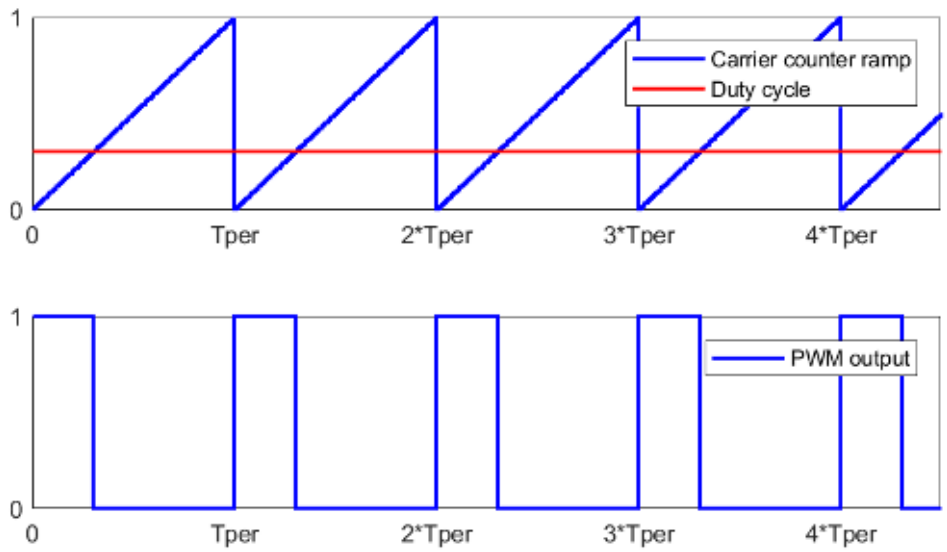
O código completo do Exemplo 2 se encontra no Apêndice C ao final deste documento.

5. Módulo *ePWM*

5.1 Introdução

O *PWM* é um tipo de modulação que funciona a partir da comparação de dois sinais, sendo um deles constante e outro variável no tempo, com isto é possível variar o ciclo de trabalho de uma onda quadrada. A Figura 5 .8mostra um exemplo de *PWM* com a comparação entre um sinal constante e uma onda dente de serra.

Figura 5.8: Onda *PWM*.



Fonte: *Mathworks*. [6]

Onde na Figura 5 .8, o sinal vermelho é a constante definida que irá ser comparada ao sinal azul de dente de serra. Quando o sinal em rampa da portadora for menor que a referência de ciclo de trabalho, a saída *PWM* está em nível lógico alto, caso contrário está em nível lógico baixo. Nota-se que caso a referência

aumente, o ciclo de trabalho aumentará, podendo assim varrer os valores de 0 a 100% de ciclo de trabalho

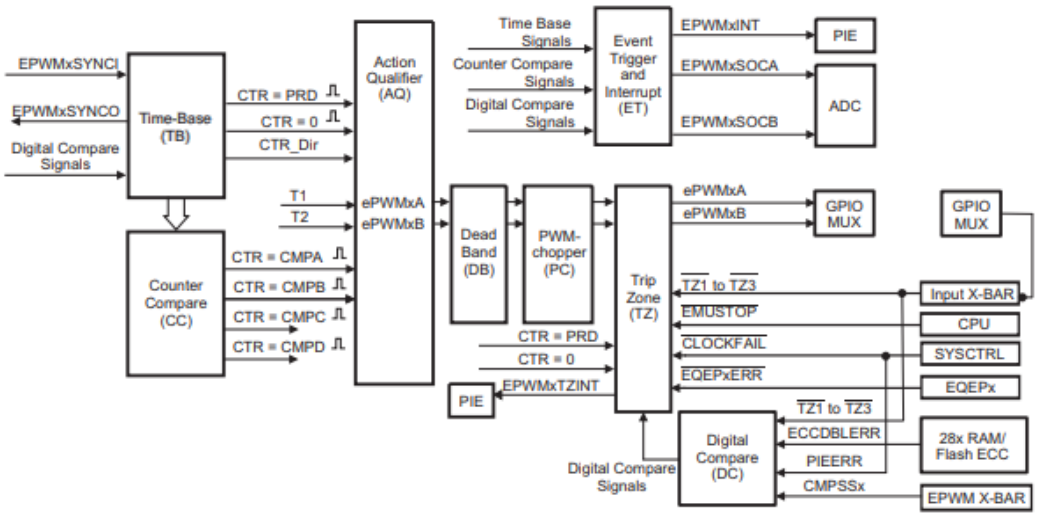
O *PWM* tem diversas aplicações em áreas da engenharia elétrica, como em telecomunicações para transferências de dados, eletrônica de potência para o chaveamento de transistores, por exemplo, em pontes H, regulação de tensão em cargas, dentre outras.

No TMS320F28377S, está presente o *ePWM*, que em comparação ao *PWM*, possui maior flexibilidade nos módulos e submódulos, e oferece uma capacidade de resolução maior também. O *ePWM* contido no TMS320F28377S contém 15 canais e é composto por outros 8 submódulos, são eles:

- *Time-Base* (*TB*);
- *Counter-Compare* (*CC*);
- *Action-Qualifier* (*AC*);
- *Dead-Band Generator* (*DB*);
- *PWM-Chopper* (*PC*);
- *Trip-Zone* (*TZ*);
- *Event-Trigger* (*ET*);
- *Digital-Compare* (*DC*).

A Figura 5 .9ilustra um diagrama em blocos destes submódulos.

Figura 5.9: Diagrama em blocos de um módulo *ePWM*.



Fonte: SPRUHX5E. [4]

5.2 Canais *ePWM*

No dispositivo tratado, temos 8 módulos *ePWM*, sendo que 7 deles possuem dois canais de saída, *EPWMxA* e *EPWMxB*, onde x representa o número do módulo. De acordo com a documentação do *Datasheet* do *DSP*, os módulos contidos são:

- *ePWM2;*
- *ePWM3* (apenas um canal);
- *ePWM6;*
- *ePWM7;*
- *ePWM8*;
- *ePWM9*;
- *ePWM10*;
- *ePWM11*.

A relação entre estes dois canais de saída A e B são definidos pelo programador ao escrever o código para o dispositivo. Estas definições ocorrem pela configuração dos submódulos. Para a utilização destes canais em pinos *GPIO*, é necessário que a multiplexação dos pinos seja feita corretamente. A documentação SPRUI25C [1] apresenta os pinos, e a multiplexação necessária para a operação do *ePWM* desejado. O *ePWM7* já tem suas duas saídas conectadas aos *LEDs* da placa (azul e vermelho), sendo assim basta que a multiplexação seja feita corretamente para que o usuário possa utilizá-los em seus testes.

Todos os códigos criados, via *MATLAB*™, para geração das formas de onda deste capítulo estão presentes no final deste documento no Apêndice G.

5.3 Modos de Contagem e Base de Tempo (*TB*)

O *ePWM*, e também o *PWM*, podem trabalhar com 3 métodos de contagem da onda portadora mostrada na Figura 5 .8, são eles:

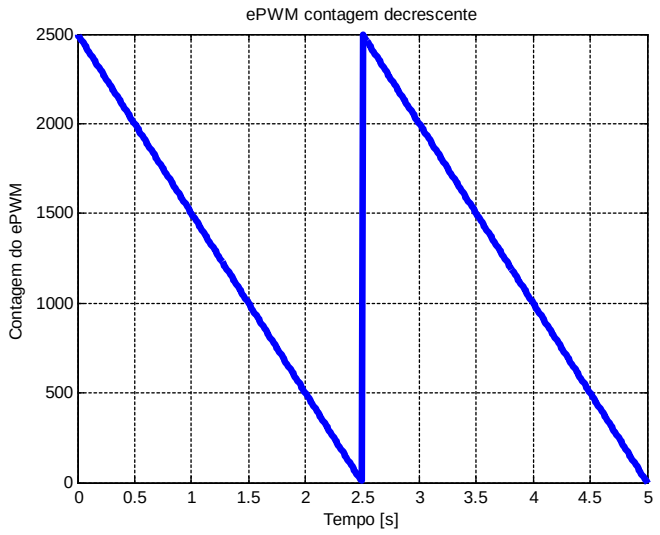
- Contagem crescente;
- Contagem decrescente;
- Contagem crescente e decrescente.

O primeiro tipo de contagem apresenta uma forma de onda da portadora idêntica à apresentada na Figura 5 .8, onde o programador estipula o valor máximo

de contagem e a frequência de incremento deste valor, e ao chegar ao valor máximo estipulado, o contador é zerado e a contagem é reiniciada. É possível fazer com que a contagem inicial ocorra em um valor diferente de zero utilizando os submódulos do *ePWM*.

Analogamente à contagem crescente, a decrescente realiza o processo de maneira inversa. Ao invés de começar a contagem do valor nulo, ela começa no valor máximo de contagem e a frequência de decremento estipula como será o decrescimento do valor contado. A forma de onda produzida por este modo está ilustrada na Figura 5 .10.

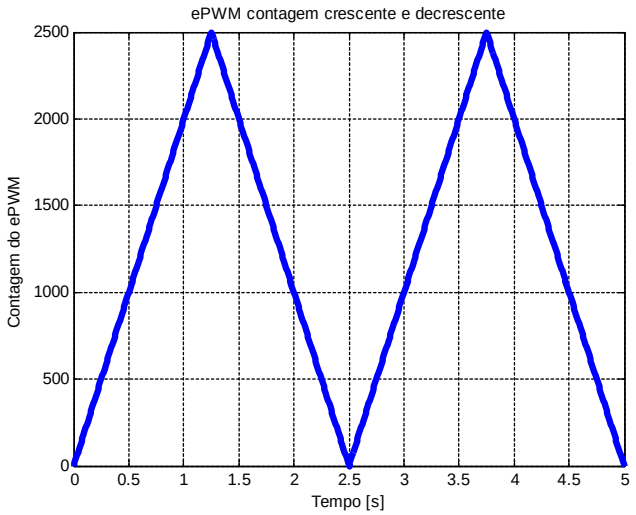
Figura 5.10: Contagem decrescente do *ePWM*.



Fonte: Autoria própria.

O último modo de contagem funciona como uma junção entre os dois anteriores, dessa forma a onda apresentada agora é um triangular, onde a contagem é inicializada de zero até o valor máximo pelo incremento constante definido pelo programador, e após isto a contagem é decrescida de mesmo valor do incremento com mesma frequência até o valor zero. A Figura 5 .11 apresenta esta forma de onda da portadora.

Figura 5.11: Contagem crescente e decrescente do *ePWM*.



Fonte: Autoria própria.

Abaixo temos alguns exemplos de como selecionar o tipo de contagem a ser executada pelo *ePWM*.

- Contagem crescente (*CTRMODE*=0x0):

```
EPwm7Regs.TBCTL.bit.CTRMODE = TB_COUNT_UP; // Seleciona a contagem crescente.
EPwm7Regs.TBCTL.bit.PHSEN = TB_DISABLE;        // Não carrega TBPHS em TBCTL.
EPwm7Regs.TBPHS.bit.TBPHS = 0x0000;            // A fase é setada como 0.
EPwm7Regs.TBCTR = 0x0000;                      // Limpa o contador.
EPwm7Regs.TBCTL.bit.HSPCLKDIV = TB_DIV1;       // SYSCLKOUT dividido por 1
EPwm7Regs.TBCTL.bit.CLKDIV = TB_DIV1;         // SYSCLKOUT dividido por 1
```

- Contagem decrescente (*CTRMODE*=0x1):

```
EPwm7Regs.TBCTL.bit.CTRMODE = TB_COUNT_DOWN; // Seleciona a contagem decrescente.
EPwm7Regs.TBCTL.bit.PHSEN = TB_DISABLE;        // Não carrega TBPHS em TBCTL.
EPwm7Regs.TBPHS.bit.TBPHS = 0x0000;            // A fase é setada como 0.
EPwm7Regs.TBCTR = 0x0000;                      // Limpa o contador.
EPwm7Regs.TBCTL.bit.HSPCLKDIV = TB_DIV1;       // SYSCLKOUT dividido por 1
EPwm7Regs.TBCTL.bit.CLKDIV = TB_DIV1;         // SYSCLKOUT dividido por 1
```

- Contagem crescente e decrescente (*CTRMODE*=0x2):

```
EPwm7Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN; //Contagem crescente e decrescente.
EPwm7Regs.TBCTL.bit.PHSEN = TB_DISABLE;        // Não carrega TBPHS em TBCTL.
EPwm7Regs.TBPHS.bit.TBPHS = 0x0000;            // A fase é setada como 0.
EPwm7Regs.TBCTR = 0x0000;                      // Limpa o contador.
EPwm7Regs.TBCTL.bit.HSPCLKDIV = TB_DIV1;       // SYSCLKOUT dividido por 1
EPwm7Regs.TBCTL.bit.CLKDIV = TB_DIV1;// SYSCLKOUT dividido por 1
```

Os valores para *CTRMODE* estão definidos como mostrado anteriormente, e o valor de 0x3 serve para congelar a contagem, o que pode ser necessário para o programador em algum ponto do código. A maioria das funções dos programas exemplo da *Texas Instruments*™ possui defines que tornam a programação mais

lógica, abaixo está uma parte dos defines existentes na biblioteca do *DSP*, presentes não código *EPwm_defines.h*.

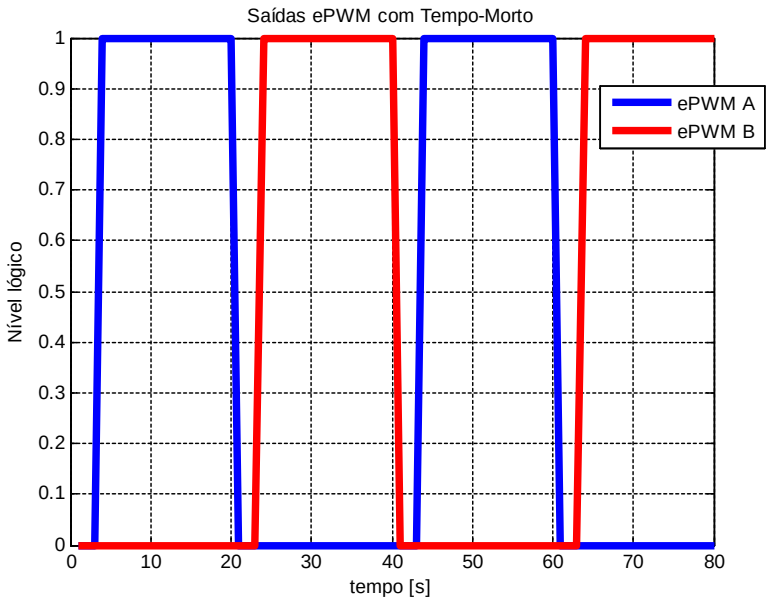
```
// CTRMODE bits
#define TB_COUNT_UP       0x0
#define TB_COUNT_DOWN     0x1
#define TB_COUNT_UPDOWN   0x2
#define TB_FREEZE         0x3
```

5.4 Tempo-Morto (*DB*)

O tempo-morto é um dos submódulos presentes no *ePWM*, e é responsável pela produção de *delays* nas saídas *PWM* para evitar, por exemplo, curtos-circuitos em pontes-H. A configuração deste submódulo é feita por meio dos registradores de controle *DBCTL* e *DBCTL2*, que são responsáveis pela maioria das operações gerais, e os registradores *DBRED* e *DBFED* utilizados para armazenar valores relacionados ao tempo-morto.

Em pontes-H, caso ambos os sinais de *PWM* que controlam a ativação das chaves estejam em alto, podem ocorrer danos à carga. Sendo assim, o tempo morto é utilizado para evitar que ambos fiquem em nível baixo antes que ocorra uma transição de nível baixo para alto. A ação do tempo-morto está ilustrada Figura 5 . 12, com um tempo morto de 3 segundos.

Figura 5.12: Tempo-morto na saída *ePWM*.



Fonte: Autoria própria.

Os modos de operação seguem como base a saída A do *ePWM*, gerando assim a construção do sinal B. Eles podem ser:

- Gerar pares de sinais com relação de tempo-morto com base no sinal A;
- Programar estes pares de sinais para operarem:
  - *Active High* (*AH*) e *Active Low* (*AL*), onde o sinal B acompanha o nível lógico de A;
  - *Active High Complementary* (*AHC*) e *Active Low Complementary* (*ALC*), em que o sinal B acompanha inversamente o nível lógico de A.
- *Delay* programável para bordas de subida (*RED*) e descida (*FED*);
- Não realizar nenhuma operação (*bypass*).

Este submódulo pode ser totalmente ignorado pelo sinal, basta que seja escrito para o *bitOUT_MODE* do registrador *DBCTL* os *bits* “00”. Caso queira que apenas o sinal A seja ignorado pelo *DB*, a escrita deve ser “01”, analogamente para o sinal B, a escrita deve ser “10”, e caso ambos os sinais devam passar pelo submódulo, a escrita deve ser “11”.

O código abaixo exemplifica como utilizar o submódulo para gerar o *delay* necessário para evitar os curtos-circuitos já citados.

```
// Registros de Delay
EPwm7Regs.DBCTL.bit.OUT_MODE = DB_FULL_ENABLE; // Habilita delay para as bordas de
subida e descida
EPwm7Regs.DBCTL.bit.POLSEL = DB_ACTV_HIC; // Controla a seleção de polaridade,
inverte o EPWM7B e funciona no complementar ativo em alto
EPwm7Regs.DBCTL.bit.IN_MODE = DBA_ALL; // Controle de bandamorta, a fonte para
ambos os delays de subida e descida é EPWM7A.
EPwm7Regs.DBRED.bit.DBRED = 50; // Delay de subida recebe o valor de EPWM7_MAX_DB
EPwm7Regs.DBFED.bit.DBFED = 50; // Delay de descida recebe o valor de EPWM7_MAX_DB
```

5.5 *Action-Qualifier* (*AQ*)

O AQ é o submódulo que rege como as formas de onda de saída do *ePWM* irão mudar com certas ações. O programador define pela escrita no registrador *AQCTLy*, onde “y” é a letra da saída *ePWM* em questão que ação será tomada quando a contagem atingir certo valor.

As ações que o submódulo *AQ* pode performar são:

- Setar, limpar ou realizar toggle nas saídas *ePWM* nos eventos:

    - Contagem no valor máximo;
    - Contagem no valor zero;
    - Contagem no valor de *CMPA*;
    - Contagem no valor de *CMPB*;
- Gerar eventos de *trigger*;
- Definir prioridades de eventos;
- Distinguir eventos que ocorrem em contagens crescentes e decrescentes.

O registrador *AQTSRCSEL* seleciona as fontes de *trigger* de eventos, que podem ser considerados via escrita do *AQCTLA2*, e configurações gerais, como operação em *shadow* do submódulo, em *AQCTL*, os quais não serão detalhados aqui, mas o leitor pode consultá-los na documentação SPRUHX5E [4].

Por exemplo, caso sejam desejadas saídas complementares A e B, o seguinte código pode ser aplicado:

```
EPwm7Regs.AQCTLA.bit.CAU = 0x1;// Seta ePWM7A quando TBCTR = CMPA
                                  // na contagem crescente
EPwm7Regs.AQCTLA.bit.CAD = 0x2;       // Limpa ePWM7A quando TBCTR =
                                  // CMPA na contagem decrescente
EPwm7Regs.AQCTLB.bit.CAD = 0x1;      // Seta ePWM7B quando TBCTR = CMPA
                                  // na contagem decrescente
EPwm7Regs.AQCTLB.bit.CAU = 0x2;       // Limpa ePWM7B quando TBCTR = CMPA
                                  // na contagem crescente
```

No registrador *AQCTLy*, 6 pares de *bits* representam 4 tipos de ações que podem acontecer para cada evento diferente. Os seis eventos são relacionados ao registro de contagem *TBCTR*, são eles:

1. *TBCTR=CMPB* na contagem decrescente (*CBD*);
2. *TBCTR=CMPB* na contagem crescente (*CBU*);
3. *TBCTR=CMPA* na contagem decrescente (*CAD*);
4. *TBCTR=CMPA* na contagem crescente (*CAU*);
5. *TBCTR=TBPRD* (*PRD*);
6. *TBCTR*=0 (*ZRO*).

O código citado anteriormente pode gerar a onda mostrada na Figura 5.5, onde a saída A e B do *ePWM* são complementares.

5.6 *Counter-Compare* (*CC*)

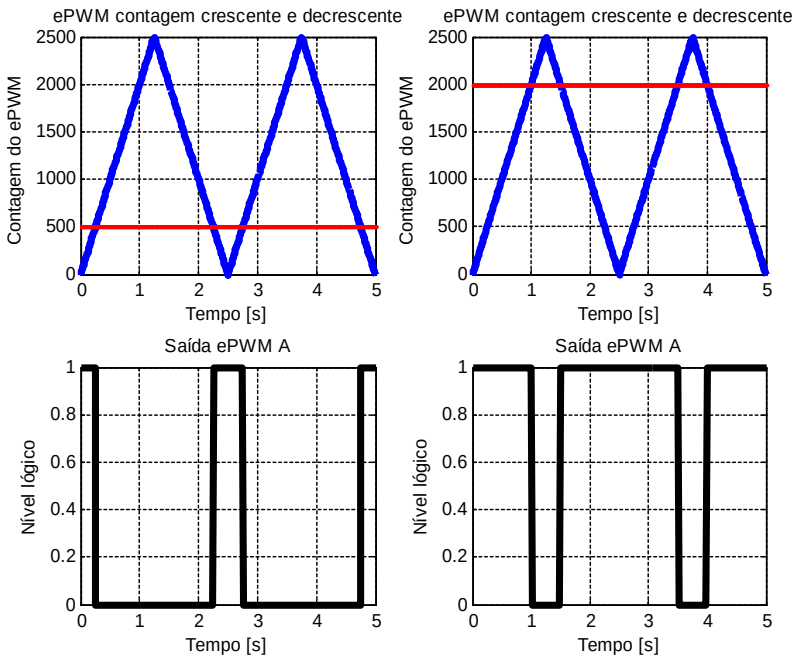
Este submódulo tem a função de gerar os pulsos do *ePWM* com base na comparação de valores entre *TBCTR* e um valor definido pelo usuário, logo, com ele é possível controlar o *duty-cicle* do *ePWM*. Os valores definidos pelo usuário são armazenados nos registradores *CMPA* e *CMPB*, eles podem operar em modo shadow dependendo da escrita nos *bits* de *SHDWyMODE*, onde "y" é a letra que representa qual comparador está sendo afetado. Estes *bits* se encontram no registrador *CMPCTL*. Para que se escolha quando carregar os valores ao registrador ativo, utiliza-se a escrita nos *bitsLOADyMODE*, também contido em *CMPCTL*. O carregamento pode acontecer na contagem máxima, zero, em ambas, ou nunca.

O submódulo *CC* pode também realizar as seguintes operações:

- Gerar eventos de acordo com valores de *CMPA* e *CMPB*;
- Controlar o *duty-cicle* do *ePWM* em aliança com o submódulo *AC*;
- Utilizar o modo *shadow* para evitar *glitches*.

A Figura 5 .13 apresenta como o ciclo de trabalho pode ser mudado com a operação em conjunto de *CC* e *AQ*. Em cada caso apresentado, o programa é o mesmo, apenas o valor de comparação que foi mudado. Na Figura 5 .13, os sinais em azul representam a contagem do *ePWM*, em vermelho o valor de comparação, e em preto a saída A do *ePWM*.

Figura 5.13: Mudança no ciclo de trabalho.



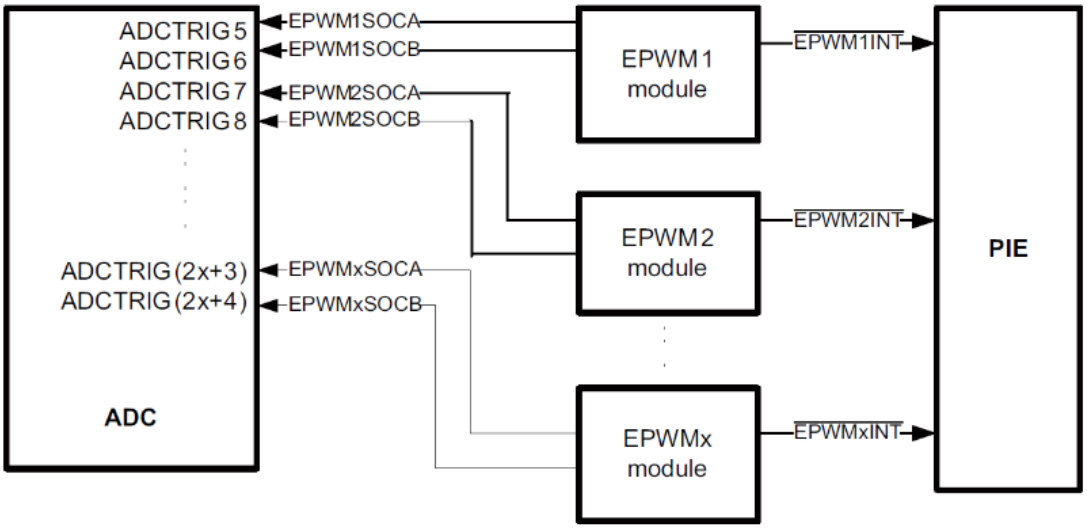
Fonte: Autoria própria.

5.7 *Event-Trigger* (*ET*)

O *Event-Trigger* é um bloco importante quando um módulo *ePWM* é utilizado para gerar interrupções na *CPU* ou como *trigger* do *ADC*. Entre suas principais funções: receber os eventos gerados pelos módulos *Time-Base, Counter-Compare* e *Digital Compare*; gerar interrupções na *CPU* ou disparar o *ADC* para todo evento, cada segundo evento ou cada terceiro evento.

Cada módulo *ePWM* está conectado à *PIE* por meio de um *bitEPWMxINT* e ao *ADC* por meio dos *bitsEPWMxSOCA* ou *EPWMxSOCB*. A Figura 5 .14 esquematiza essas conexões.

Figura 5.14: Conexões dos *bits* do módulo *Event-trigger*.



Fonte: SPRUHX5E. [4]

O submódulo é configurado para gerar um pulso, dado um tipo de evento entre os listados abaixo:

- *TBCTR* = 0;
- *TBCTR* = *PRD*;
- *TBCTR* = 0 ou *TBCTR* = PRD;
- *TBCTR* = *CMPA*, quando o contador está sendo incrementado;
- *TBCTR* = *CMPA*, quando o contador está sendo decrementado;
- *TBCTR* = *CMPB*, quando o contador está sendo incrementado;
- *TBCTR* = *CMPA*, quando o contador está sendo decrementado.

5.8 Exemplo – *ADC* com *SOC* dado pelo *ePWM*

Este exemplo irá apresentar como o *ADC* pode receber um *SOC* via *ePWM*. Este programa tem interdependência direta com as explicações dadas no capítulo 4 sobre *ADC*. O programa consiste em duas entradas *ADC*, as quais serão responsáveis pelo valor de *CMPA*. O valor das duas serão convertidos via *ADC*, e a média delas enviadas para o comparador. Ambas as entradas *ADC* foram ligadas em 3.3 V CC. Os valores dos ciclos de trabalho serão representados pelo brilho dos dois *LEDs* da *Launchpad*™. A ideia contida é de que as saídas *PWM* estarão invertidas, sendo assim o ciclo de trabalho máximo para uma entrada significa o mínimo para a saída complementar, visualmente isto acarretará em variar o brilho dos *LEDs*.

Primeiramente serão declaradas as variáveis globais, funções protótipo e includes necessários.

```
#include"F28x_Project.h"
// Funções utilizadas no programa
void ConfigureADC(void);
void ConfigureEPWM(void);
void SetupADCEpwm(void);
interrupt void adca1_isr(void);
__interrupt void epwm7_isr(void);
#define EPWM7_TIMER_TBPRD 1628 // Contagem máxima de 1628 para termos uma
                               // frequência próxima de 15360 Hz
#define EPWM7_MAX_CMPA     814 // Comparador A máximo
#define EPWM7_MAX_CMPB     814 // Comparador B máximo
#define EPWM7_MAX_DB        50 // Dead-Band máxima
// Variáveis globais do programa
float32 media=EPWM7_MAX_CMPA;
```

Onde média será a variável responsável pela média das conversões *AD* das duas entradas. Será feito a inicialização da *main* assim como já descrito na seção 3.1. porém com ligeiras mudanças para priorizar a utilização da *Launchpad*™ do *DSP*.

```
void main(void)
{
InitSysCtrl(); // Comentado.
InitGpio(); // Comentado.
CpuSysRegs.PCLKCR2.bit.EPWM7=1; // Peripheral Clock Gating Registers, habilita
                                // o ePWM7.
InitEPwm7Gpio(); // Comentado.
```

Será utilizado o *ePWM7* para o exemplo por conta de ser o canal que tem a interligação do mesmo com os *LEDs* da *Launchpad*™, o que facilita a visualização da mudança do ciclo de trabalho nos mesmos.

```
EALLOW;
```

```
GpioCtrlRegs.GPAMUX1.bit.GPIO13 = 0x00;  // Evitar glitch (LED AZUL).
GpioCtrlRegs.GPAGMUX1.bit.GPIO13 = 0x00; // Mux para EPWM7B estar no led azul.
GpioCtrlRegs.GPAMUX1.bit.GPIO13 = 0x01;  // Mux para EPWM7B estar no led azul.
GpioCtrlRegs.GPAMUX1.bit.GPIO12 = 0x00;  // Evitar glitch (LED VERMELHO).
GpioCtrlRegs.GPAGMUX1.bit.GPIO12 = 0x00; // Mux para EPWM7A estar no led
                                         // vermelho.
GpioCtrlRegs.GPAMUX1.bit.GPIO12 = 0x01;  // Mux para EPWM7A estar no led
                                       // vermelho.
GpioCtrlRegs.GPAPUD.bit.GPIO13 = 1;      // Desabilita pullup.
GpioCtrlRegs.GPAPUD.bit.GPIO12 = 1;      // Desabilita pullup.
GpioCtrlRegs.GPADIR.bit.GPIO13 = 1;      // Define como saída.
GpioCtrlRegs.GPADIR.bit.GPIO12 = 1;      // Define como saída.
EDIS;
```

Apenas a configuração usual para evitar *glitches* nos *GPIOs* e garantir que as saídas estejam conectadas com a correta multiplexação dos pinos.

```
DINT; // Desabilita interrupções.
InitPieCtrl(); // Comentado.
IER = 0x0000; // Desabilita interrupções da CPU.
IFR = 0x0000; // Limpa flags de interrupção.
InitPieVectTable(); // Comentado.
EALLOW;
PieVectTable.ADCA1_INT = &adca1_isr; // Função para interrupção do ADCA
                                     // linkada com a tabela PIE.
PieVectTable.EPWM7_INT = &epwm7_isr; // Função para interrupção do ePWM7
                                     // linkada com a tabela PIE.
EDIS;
```

Aqui ocorreu o mapeamento das duas funções de interrupções, uma do *ADC* para calcular a média das entradas convertidas, e a do *ePWM*, a qual será responsável por escrever o novo valor de *CMPA* no registro em *shadow*.

```
ConfigureADC(); // Inicializa o ADCA e o energiza.
```

Será feita agora a inicialização do *ADC* e energização do mesmo de maneira semelhante a feita em exemplos anteriores.

```
// ConfigureADC - Escreve as configurações e inicialização do ADC.
void ConfigureADC(void)
{
EALLOW;
AdcaRegs.ADCCTL2.bit.PRESCALE = 0;     // Divide ADCCLK por 1.
// Inicializa o ADCA com uma resolução de 12 bits no modo de sinal Single Ended.
AdcSetMode(ADC_ADCA, ADC_RESOLUTION_12BIT, ADC_SIGNALMODE_SINGLE);
AdcaRegs.ADCCTL1.bit.INTPULSEPOS = 1; // Faz com que o pulso de interrupção
                                      // ocorra ao final de conversão (EOC) do AD.
AdcaRegs.ADCCTL1.bit.ADCPWDNZ = 1;   // Energiza por completo o módulo ADCA.
DELAY_US(1000);  // Delay de 1 ms para permitir a correta energização do ADC.
EDIS;
}
```

Foi feito com que o *clock* do *ADC* não fosse alterado de seu valor *default* de 100 MHz, e a inicialização ocorresse para o *ADC* A, com a resolução de 12 *bits* e a

configuração do mesmo em *single-ended*. Logo após isto foi setado que o pulso de interrupção seria dado ao final de conversão (*EOC*), pois para a aplicação é interessante que a interrupção ocorra ao final de conversão, e por fim ocorre a inicialização do módulo. Voltando à *main*, será garantida o congelamento de contagem do *ePWM* e chamada a função de configuração do canal do *ePWM*.

```
EALLOW;
CpuSysRegs.PCLKCR0.bit.TBCLKSYNC = 0; // Mantém a contagem parada dos módulos
                                      // ePWM.
EDIS;
ConfigureEPWM(); // Configura o ePWM7
```

A função *ConfigureEPWM* irá configurar os submódulos necessários do canal para a aplicação.

```
void ConfigureEPWM(void)
{
EALLOW;
// Registros de contagem.
EPwm7Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN; // Seleciona a contagem
// crescente e decrescente, ou seja, triangular.
EPwm7Regs.TBCTL.bit.PHSEN = TB_DISABLE;        // Não carrega TBPHS em TBCTL.
EPwm7Regs.TBPHS.bit.TBPHS = 0x0000;            // A fase é setada como 0.
EPwm7Regs.TBCTR = 0x0000;                      // Limpa o contador.
EPwm7Regs.TBCTL.bit.HSPCLKDIV = TB_DIV1;       // Clock ratio to SYSCLKOUT
EPwm7Regs.TBCTL.bit.CLKDIV = TB_DIV1;
```

As primeiras configurações realizadas do *ePWM* são no submódulo do *TB*, onde ele primeiramente seta a contagem como crescente e decrescente, desabilita o carregamento de fase, e garante que a mesma é 0. Após isto, o registro de contagem é setado como zero, e a taxa do *clock* é colocada como a mesma de *SYSCLKOUT*.

```
// Registros de Delay
EPwm7Regs.DBCTL.bit.OUT_MODE = DB_FULL_ENABLE; // Habilita delay para as
// bordas de subida e descida
EPwm7Regs.DBCTL.bit.POLSEL = DB_ACTV_HIC;      // Controla a seleção de
// polaridade, inverte o EPWM7B e funciona no complementar ativo em alto
EPwm7Regs.DBCTL.bit.IN_MODE = DBA_ALL;         // Controle de banda morta, a
                    // fonte para ambos os delays de subida e descida é EPWM7A.
EPwm7Regs.DBRED.bit.DBRED = EPWM7_MAX_DB;      // Delay de subida recebe o
                                               // valor de EPWM7_MAX_DB
EPwm7Regs.DBFED.bit.DBFED = EPWM7_MAX_DB;      // Delay de descida recebe o
                                               // valor de EPWM7_MAX_DB
```

Os registros de *delay*, ou tempo-morto estão para garantir que as saídas A e B nunca estejam em nível alto ao mesmo tempo. O valor de *EPWM7_MAX_DB* já foi

especificado para ser respectivo a 1 µs. É selecionado primeiramente então que o tempo-morto atuará em bordas de subida e descida, e que a fonte deste será *EPWM7A*. A seleção de polaridade é tal que para *EPWM7B* é lido como o complementar ativo em alto.

```
// Registros de Shadow
EPwm7Regs.CMPCTL.bit.SHDWAMODE = CC_SHADOW;     // Ativa o modo Shadow para o
                                                // comparador A
EPwm7Regs.CMPCTL.bit.SHDWBMODE = CC_SHADOW;     // Ativa o modo Shadow para o
                                                // comparador B
EPwm7Regs.CMPCTL.bit.LOADAMODE = CC_CTR_PRD;    // Ativa o carregamento do
            // valor de Shadow A quando o contador atinge a contagem de PRD
EPwm7Regs.CMPCTL.bit.LOADBMODE = CC_CTR_PRD;    // Ativa o carregamento do
            // valor de Shadow B quando o contador atinge a contagem de PRD
```

A configuração nos registros de *shadow* para os comparadores é feita para que o valor de comparação seja atualizado no registro efetivamente apenas quando a contagem de *TBCTR* alcance o valor de *PRD*.

```
// Registros de Comparação e Ação
EPwm7Regs.CMPA.bit.CMPA = EPWM7_MAX_CMPA;     // Valor de comparação A
EPwm7Regs.CMPB.bit.CMPB = EPWM7_MAX_CMPB;    // Valor de comparação B
EPwm7Regs.AQCTLA.bit.CAU = AQ_CLEAR;         // Seta ePWM7A quando TBCTR =
                                             // CMPA na contagem crescente
EPwm7Regs.AQCTLA.bit.CAD = AQ_SET;            // Limpa ePWM7A quando TBCTR =
                                              // CMPA na contagem decrescente
EPwm7Regs.AQCTLB.bit.CAD = AQ_CLEAR;          // Seta ePWM7B quando TBCTR =
                                              // CMPA na contagem decrescente
EPwm7Regs.AQCTLB.bit.CAU = AQ_SET;            // Limpa ePWM7B quando TBCTR =
                                              // CMPA na contagem crescente
// Desta forma, os valores de ePWM7A e ePWM7B são complementares SEMPRE !!
```

Foi programado então que as saídas A e B serão complementares sempre, desta maneira, todas as configurações anteriores para a saída B se tornam dispensáveis. Elas foram apresentadas até aqui apenas com o caráter didático.

```
// Registros de interrupção
EPwm7Regs.ETSEL.bit.INTSEL = ET_CTR_PRD;        // Interrupção ocorre quando
                                                // TBCTR = TBPRD
EPwm7Regs.ETSEL.bit.INTEN = 1;                  // Habilita a interrupção
EPwm7Regs.ETPS.bit.INTPRD = ET_1ST;             // Gera a interrupção sempre no
                                                // 1º evento
```

As interrupções do *ePWM* são configuradas para ocorrerem quando a contagem alcança o valor de *PRD*, e ela acontece sempre no 1º evento, ou seja, a cada pico de *TBCTR*, haverá a interrupção.

```
// Registros de SOC
EPwm7Regs.ETSEL.bit.SOCAEN = 0;          // Desabilita SOC no grupo A
EPwm7Regs.ETSEL.bit.SOCASEL = 2;         // Dispara o SOC quando TBCTR = TBPRD
EPwm7Regs.ETPS.bit.SOCAPRD = 1;          // Gera pulso de SOC no 1º evento
EPwm7Regs.TBPRD = EPWM7_TIMER_TBPRD; // Seta o período com o valor já definido
EPwm7Regs.TBCTL.bit.CTRMODE = 3;                // Congela o contador
EDIS;
}
```

Por fim, os registros de *SOC* irão configurar que o início de conversão sempre ocorre no pico de contagem, ou seja, *TBCTR=TBPRD*, e este *SOC* sempre ocorrerá no primeiro evento. Após isto apenas é setado o valor de *TBPRD* como o valor pré-definido e o congelamento do contador. Voltemos à *main*.

```
EALLOW;
CpuSysRegs.PCLKCR0.bit.TBCLKSYNC = 1; // Permite a contagem dos módulos ePWM.
EDIS;
SetupADCEpwm(); // Inicializa o ADCA canal 0 e 1
```

De volta à *main*, é agora permitida a contagem dos módulos *ePWM*, e chamada a função de inicialização do canal 0 e 1 de *ADCA* em conjunto com o *ePWM*.

```
void SetupADCEpwm(void)
{
Uint16 acqps;
if(ADC_RESOLUTION_12BIT == AdcaRegs.ADCCTL2.bit.RESOLUTION)
// Testa a resolução, em bits, da conversão AD, caso seja 12 bits segue o if
// caso contrário, executa o else, com a resolução de 16 bits.
{
acqps = 14; //75 ns
}
else//resolution is 16-bit
{
acqps = 63; //320 ns
}
EALLOW;
AdcaRegs.ADCSOC0CTL.bit.CHSEL = 0; // SOC0 irá converter o pino ADCINA0
AdcaRegs.ADCSOC0CTL.bit.ACQPS = acqps;      // Atribui a janela de amostragem
                            // o valor relativo a resolução pré-estabelecida.
AdcaRegs.ADCSOC0CTL.bit.TRIGSEL = 0x11;// Trigger em SOCA0 a partir do ePWM7
// Adição de inicialização também do canal 1 de ADCA
AdcaRegs.ADCSOC1CTL.bit.CHSEL = 1;    // SOC1 irá converter o pino ADCINA1
AdcaRegs.ADCSOC1CTL.bit.ACQPS = acqps;      // Atribui a janela de amostragem
                             // o valor relativo a resolução pré-estabelecida.
AdcaRegs.ADCSOC1CTL.bit.TRIGSEL = 0x11;// Trigger em SOCA1 a partir do ePWM7
AdcaRegs.ADCINTSEL1N2.bit.INT1SEL = 1; // O fim de SOC1 irá setar o flag INT1
AdcaRegs.ADCINTSEL1N2.bit.INT1E = 1;        // Habilita o flag de INT1
AdcaRegs.ADCINTFLGCLR.bit.ADCINT1 = 1;// Garante o flag de INT1 limpo
EDIS;
}
```

Primeiramente é escolhido o número de ciclos de *SYSCLK* necessários para a conversão de acordo com a resolução do *ADC*. Depois este valor vai ser armazenado nos registradores dos canais 0 e 1, e para ambos os canais é escolhido *ePWM7* como fonte de *trigger* de *SOCA0* e *SOCA1*. O final da conversão do canal 1 que irá setar o *flag* de *INT1*, que é habilitado e limpo logo após. Com isto setado já é possível que ocorra a interrupção devido ao *ADC*. Voltemos à *main*.

```
IER |= M_INT1; // Habilita o grupo 1 de interrupções, serve para o ADC
IER |= M_INT3; // Habilita o grupo 3 de interrupções, serve para o ePWM
EINT;          // Habilita as interrupções globais.
```

```
ERTM;          // Habilita as interrupções de tempo real.
```

Os *enables* de interrupções dos grupos 1 e 3 referentes, respectivamente, ao *ADC* e *ePWM* são setados para que tais interrupções sejam reconhecidas pelo dispositivo. Logo após, as interrupções globais e de tempo real são habilitadas.

Logo após, serão habilitadas as interrupções de *ADCA* e *ePWM7* na *PIE*, e inicializada a contagem dos módulos *ePWM*.

```
PieCtrlRegs.PIEIER1.bit.INTx1 = 1; // Habilita interupção do ADCA
PieCtrlRegs.PIEIER3.bit.INTx7 = 1; // Habilita interrupção do ePWM7
EALLOW;
CpuSysRegs.PCLKCR0.bit.TBCLKSYNC = 1; // Permite a contagem dos módulos ePWM.
EDIS;
```

Para fim, será habilitado o *SOCA*, o descongelamento da contagem e a entrada no *loop* infinito.

```
EPwm7Regs.ETSEL.bit.SOCAEN = 1;  // Habilita o SOCA
EPwm7Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN; // descongela a contagem
do
{
}while(1);
}
```

As rotinas de tratamento de interrupção serão tratadas agora. A do *ePWM7* ocorre primeiro, sendo assim iremos detalhá-la primeiramente.

```
__interrupt void epwm7_isr(void)
{
EPwm7Regs.CMPA.bit.CMPA = media; // Escreve o novo valor de CMPA de acordo
                                 // com a última conversão condicionada
EPwm7Regs.ETCLR.bit.INT = 1;             // Limpa o flag de interrupção
PieCtrlRegs.PIEACK.all = PIEACK_GROUP3;  // Permite que outras interrupções
                                         // da natureza do Grupo 3 ocorram novamente
}
```

O valor da média é atribuído ao registro de comparação, para que a partir deste momento, o ciclo de trabalho das saídas A e B mudem. O *flag* de interrupção é limpo e o comando *PIEACK* faz com que as próximas interrupções desta fonte ocorram novamente após um novo evento. A interrupção próxima é a do *ADCA*.

```
interrupt void adca1_isr(void)
{
media = 0.19875*(AdcaResultRegs.ADCRESULT0 + AdcaResultRegs.ADCRESULT1);
AdcaRegs.ADCINTFLGCLR.bit.ADCINT1 = 1;  // Limpa o flag de interrupção
PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;// Permite que outras interrupções
                                       // da natureza do Grupo 1 ocorram novamente
}
```

Com a interrupção de *ADCA*, o valor da média é calculado. O valor 0,19875 é um fator de correção para adequar os valores máximos que as conversões geram ao valor máximo de *TBPRD*. Os dois comandos a seguir têm a mesma função que na interrupção do *ePWM7*, porém agora voltados para o *ADCA*.

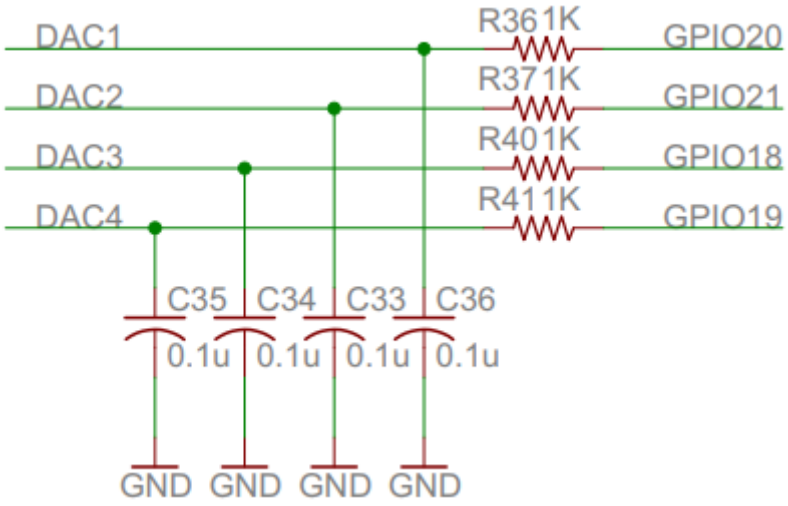
O programa condensado está disponível no Apêndice D deste documento. Uma implementação semelhante à esta, porém com a utilização de dois canais em *ADC* A e outros dois em *ADC* B, e 3 *ePWMs* diferentes está presente no Apêndice E deste documento

6. *Buffered Digital-to-Analog Converter* (*DAC*)

7. Introdução

O conversor digital-analógico funciona na via oposta do *ADC*, convertendo um sinal digital estipulado pelo programador no código para um valor correspondente analógico. No F28377S, existem 3 *DACs* de 12 *bits*, e outros 4 implementados via *ePWM* com um filtro passa-baixa implementado. Os *DACs* de 12 *bits* estão presentes nos mesmos pinos de *ADCINA0*, *ADCINA1* e *ADCINB1*, enquanto os outros 4, possuem pinos específicos para utilização do programador. Tais filtros estão ilustrados na Figura 6 .15.

Figura 6.15: Implementação do *DAC* com filtro.



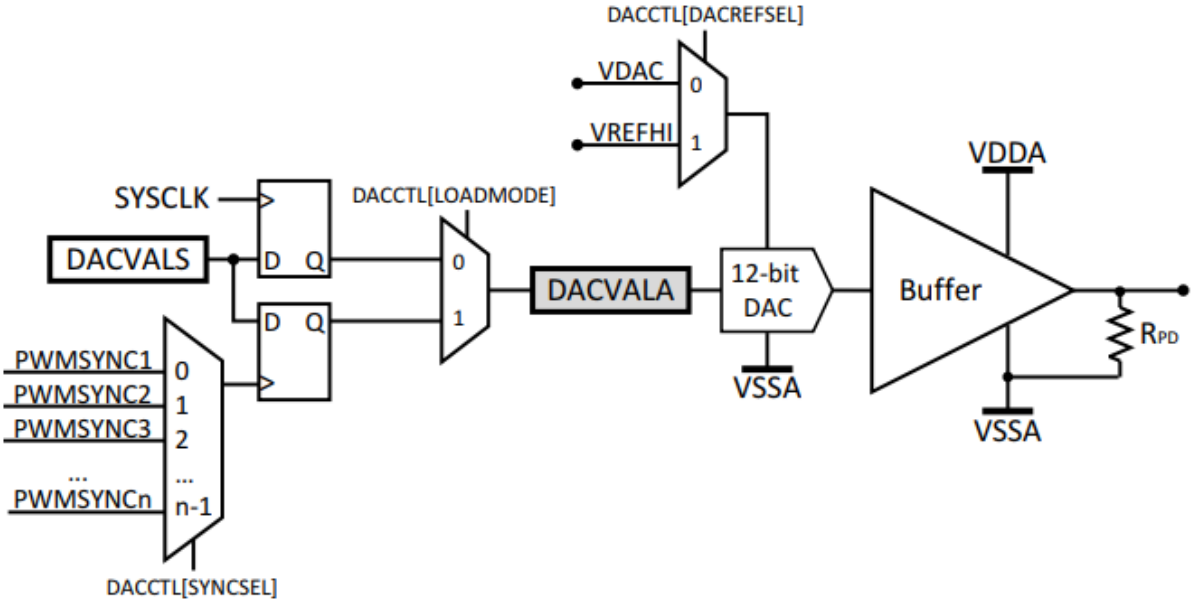
Fonte: SPRUI25C. [1]

8. Funcionamento

Aqui será explicado o funcionamento do *DAC* de 12 *bits* presente no F28377S. A Figura 6 .16 apresenta um diagrama de blocos do funcionamento do *DAC*. O valor selecionado pelo usuário para a saída do *DAC* está contido em *DACVALS*, que é um registro em *shadow*. A atualização do valor para o registro ativo *DACVALA* pode ser feito em sincronia com *SYSCLK* ou um dos *ePWMs*, que é escolhido de acordo com a escrita do registro *DACCTL* nos *bits* de *SYNCSEL*. A seleção da sincronia é feita de acordo com o registro *DACCTL*, na escrita dos *bits* de *LOADMODE*.

Estas configurações permitem que o valor escrito pelo usuário chegue ao registro ativo *DACVALA*. Por fim, é necessário selecionar qual a referência de tensão positiva para o *DAC*, que pode ser a interna ao dispositivo, *VREFHI*, ou uma externa *VDAC*. Esta seleção é feita também pelo registro DACCTL, e na escrita dos *bits* de *DACREFSEL*. Com isto o *DAC* está totalmente configurado para a sua operação.

Figura 6.16: Diagrama de blocos do *DAC*.



Fonte: SPRS811F. [5]

O *buffer* na saída tem função de evitar efeitos de cargas entre o circuito posterior e o de sua entrada. O resistor R_{PD} tem função de *pull-down* para o pino, pois o *DAC* de 12 *bits* está presente no mesmo pino de outras funções, as quais são multiplexadas analogicamente no F28377S. Desta maneira esta resistência é inerte a este pino.

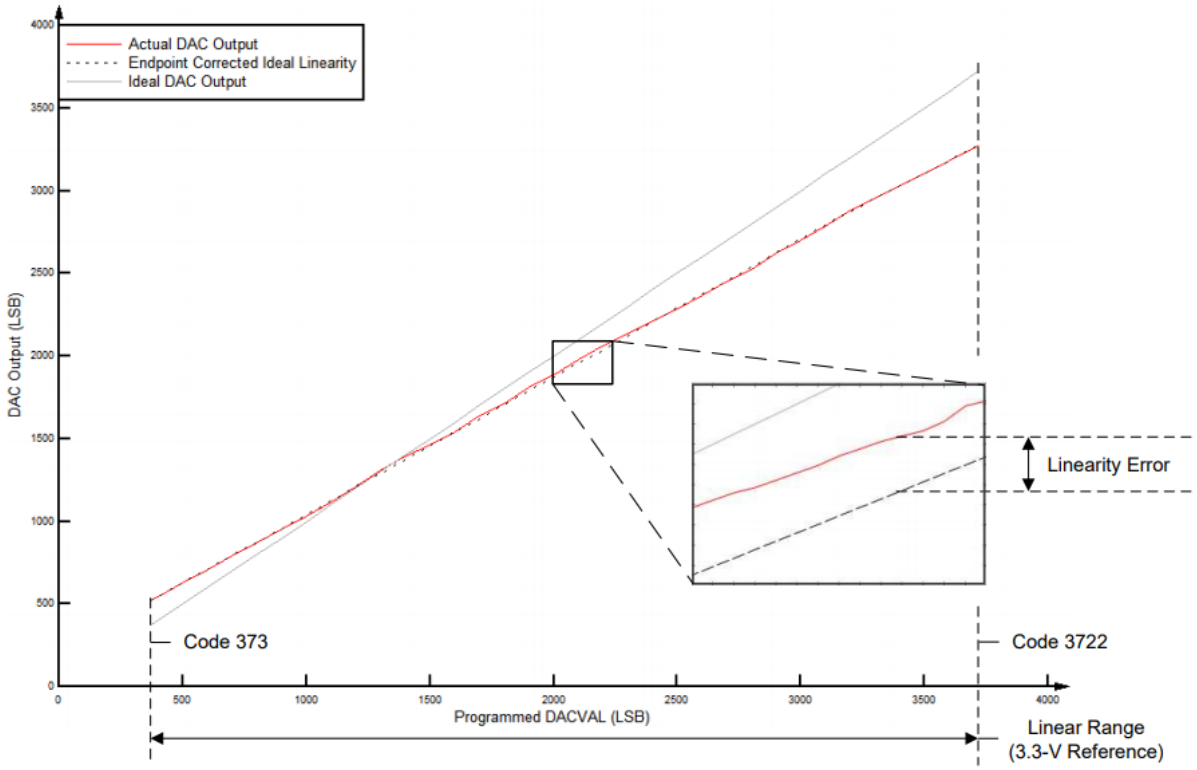
9. *DAC* em operação

Ao operar o *DAC*, a tensão de saída pode ser calculada por (6 .1) [4]:

$$V_{DAC} = \frac{DACVALA \cdot DACREF}{4096} . ¿6.1¿$$

Porém no *DSP* em trabalho, assim como em qualquer outro, é necessário respeitar a faixa linear de operação do mesmo. Além disto tanto o ganho quanto o *offset* apresentado não são iguais aos calculados no projeto do *DAC*. A Figura 6 .17 apresenta a faixa linear em que o *DAC* consegue responder satisfatoriamente. Caso seja de interesse do leitor a análise dos gráficos de *offset* e ganho do *DAC*, os mesmos se encontram no datasheet do *DSP*.

Figura 6.17: Linearidade do *DAC*.



Fonte: SPRS811F. [5]

Vê-se que a faixa linear se encontra entre 3722 e 373, ao se tratar uma referência de tensão positiva contínua de 3,3 V. O que corresponde à uma faixa de tensão entre aproximadamente 0,3005 V e 2,9987 V, que corresponde ao que é citado no *datasheet* sobre a saída apresentar a saída entre 0,3 V e V_{DDA}-0,3 V. [5]

10. Biblioteca *SGEN*

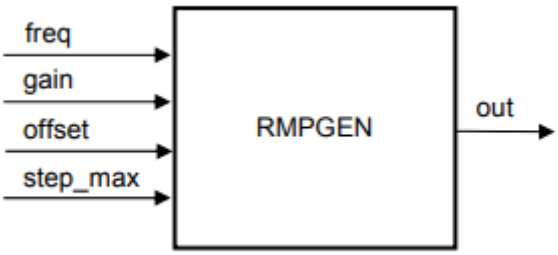
Para utilizar o *DAC* como gerador de sinais, usa-se a biblioteca *SGEN*, a qual já possui definições de variáveis e constantes para uma maior facilidade de sua utilização. Aqui será apresentada como exemplo, uma das sub-bibliotecas *SGEN*, a *RMPGEN*, utilizada para a geração de um sinal de rampa. [7]

Caso o usuário tenha a necessidade de utilizar tal biblioteca, cabe ao mesmo a consulta à bibliografia disponível em [7]. Nesta seção apresentaremos apenas a biblioteca, porém não será feito aqui um exemplo da mesma devido ao escopo proposto deste documento.

A sub-biblioteca *RMPGEN* possui quatro parâmetros de entrada, mostrados na Figura 6 .18, em que *freq* é a frequência do sinal de saída, *gain* é o ganho do sinal, *offset* é o deslocamento CC da onda, *step_max* que é a frequência máxima utilizada para a modulação do *DAC* para geração do sinal de saída e *alpha* é a fase inicial do sinal de saída. A saída de *RMPGEN* é *out* e serve apenas para disponibilizar o sinal construído pelo *DAC*.

A Tabela 6 .1apresenta estas entradas e seus respectivos ranges em numeração hexadecimal.

Figura 6.18: Bloco simplificado da *RMPGEN*.



Fonte: *SGEN Generator Library.* [7]

Tabela 6.1: Entradas e saídas da *RMPGEN*.

Item	Name	Description	Format	Range(Hex)
Input	freq	Frequency in hertz between $[-F_{MAX}, F_{MAX}]$ normalized to $[-1,1]$. The positive frequency input generates ramp up (+ve Ramp) and negative frequency input generates ramp down output (-ve Ramp)	Q15	8000-7FFF
	offset	DC offset in the ramp signal	Q15	8000-7FFF
	alpha	Initial phase $[0,2\pi]$	Q16	0-FFFF
	gain	Gain of the ramp signal	Q15	0-7FFF
	step_max	$F_{MAX} = \frac{step_\max \times F_S}{65536}$. The default value is set to 1000 to generate the maximum frequency of 305.17Hz using 20KHz sampling loop.	Q0	0000-7FFF
Output	out	SIN Output	Q15	8000-7FFF

Fonte: *SGEN Generator Library*. [7]

Todas as sub-bibliotecas *SGEN* seguem este mesmo padrão, e podem ser utilizadas para a geração de outros diversos sinais, como senóides, ondas trapezoidais, senóides trifásicas, dentre outras.

11. Exemplo – Geração de um sinal CC

Este exemplo apresenta a utilização do *DAC* para geração de um sinal de saída contínuo num valor pré-estabelecido. Primeiramente haverá a declaração de variáveis globais, inclusão de bibliotecas e definição de constantes utilizadas ao longo do código e funções protótipo.

```
// Inclusão de bibliotecas
#include "F28x_Project.h"
// Variáveis globais
volatile struct DAC_REGS* DAC_PTR[4] = {0x0,&DacaRegs,&DacbRegs,&DaccRegs};
Uint16 dacval = 4096;
// Defines
#define REFERENCE_VDAC      0
#define REFERENCE_VREF      1
#define DACA                1
#define DACB                2
#define DACC                3
#define REFERENCE           REFERENCE_VREF
#define DAC_NUM             DACA
// Funções protótipo
void configureDAC(Uint16 dac_num);
```

Diferentemente dos exemplos de outros capítulos, aqui iremos trabalhar com um ponteiro para acesso nos registros dos *DACs.* A tensão de referência utilizada será a interna de 3,3 Volts, e apenas o *DAC A*. Os demais *defines* apresentam apenas a flexibilidade possível com este código de trabalhar com mais de um *DAC*. Seguindo para escrita da função *main* a inicialização terá o padrão semelhante ao citado na Seção 3.1 deste documento.

```
void main(void)
{
InitSysCtrl();
DINT;
InitPieCtrl();
IER = 0x0000;
IFR = 0x0000;
InitPieVectTable();
configureDAC(DAC_NUM);
```

“Entrando" na função *configureDAC* temos o seguinte código:

```
// configureDAC – Configura a saída de um dado DAC
void configureDAC(Uint16 dac_num)
{
EALLOW;
//DacaRegs.DACCTL.bit.DACREFSEL = REFERENCE;
//DacaRegs.DACOUTEN.bit.DACOUTEN = 1;
//DacaRegs.DACVALS.all = 0;
DAC_PTR[dac_num]->DACCTL.bit.DACREFSEL = REFERENCE;
DAC_PTR[dac_num]->DACOUTEN.bit.DACOUTEN = 1;
DAC_PTR[dac_num]->DACVALS.all = 0;
DELAY_US(10); // Delay para inicialização correta do DAC
EDIS;
```

```
}
```

O único parâmetro desta função é o *DAC* o qual será configurado. No programa exemplo é o *DAC A*. Como os registros são protegidos, deve-se utilizar a função *EALLOW* para permitir a escrita em seus *bits*. Os comandos comentados serão explicados posteriormente.

Seguindo com o código, o primeiro comando do ponteiro *DAC_PTR*, que aponta para o endereço de *DACA,* define qual é a referência de tensão que será utilizada na geração da saída, no caso será a tensão interna de 3,3 Volts. Após isto ele habilita a saída do *DAC* e zera a saída com o último comando do ponteiro.

Por fim, o *delay* de 10 µs existe apenas para garantir a inicialização completa do *DAC* sem que antes ele seja requisitado. Após a configuração destes *bits*, para evitar novas escritas indesejadas, o comando *EDIS* é necessário.

Os comandos comentados logo após o comando *EALLOW* podem substituir os três comandos realizados por *DAC_PTR*. Caso o programador tenha a preferência por não utilizar o ponteiro, basta descomentar os três comandos e comentar os que utilizam o ponteiro.

Voltando para o código principal da *main*, temos já o final do código.

```
while(1)
{
DAC_PTR[DAC_NUM]->DACVALS.all = dacval;
//DacaRegs.DACVALS.all = dacval;
DELAY_US(2);
}
}
```

Com todas as definições já realizadas, o código entra num *loop* infinito criado pelo comando *while(1),* que dentro de suas diretrizes utilizando o comando via ponteiro, transporta o valor definido pelo usuário, em *dacval* para a saída. Para que esta tensão consiga ser entregue efetivamente na saída, existe um pequeno *delay* de 2 µs. Assim como dentro do código de *configureDAC*, caso o usuário tenha a preferência pela não utilização dos comandos via ponteiro, basta que o mesmo descomente o código comentado e comente o que utiliza os comandos via ponteiro. Este código completo para cópia do leitor está presente no final deste documento no Apêndice F.

12. Conclusão

O TMS320F28377S apresenta diversos recursos e periféricos robustos o suficiente para aplicações que demandam alta velocidade de resposta, o que necessariamente impacta numa rápida aquisição de dados.

Durante a elaboração deste documento, este dispositivo saiu de linha de acordo com a *Texas Instruments*™, e a mesma aconselhou a utilização de outro *DSP* da mesma família, o TMS320F28379D. As diferenças entre as duas plataformas não impedem que os mesmos códigos possam ser utilizados nos dois *DSPs,* com pequenas mudanças devido à localização de pinos e acesso à periféricos. Porém todas as multiplexações e *GPIOs* possuem mesma numeração, o que facilita bastante a migração de um dispositivo para o outro. O Volume II desta apostila irá tratar de outros blocos deste DSP que são essenciais para a utilização e aplicação do mesmo nos processos em que ele possa ser implementado.

Com a finalização deste tutorial, espera-se que o leitor possa ter compreendido o básico da utilização dos periféricos apresentados com o auxílio dos programas exemplo, e com consultas regulares às documentações apresentadas na bibliografia.

Referências Bibliográficas

[1] SPRUI25C – *LaunchXL – F28377S Overview, User's Guide.*Jun. 2015, revisada em Abr. de 2017. p.6, 13.

Disponível em:

www.ti.com/lit/ug/sprui25c/sprui25c.pdf

Acesso em 07/12/2017 às 15h13.

[2] *Code Composer Studio*™.

Disponível em:

www.ti.com/tool/ccstudio

Acesso em 07/12/2017 às 15h43.

[3] *ControlSUITE*™.

Disponível em:

www.ti.com/tool/CONTROLSUITE

Acesso em 07/12/2017 às 15h46.

[4] SPRUHX5E – TMS320F2837xS *Delfino Microcontrollers – Technical Reference Manual.* Ago. 2014, revisada em Set. 2017. p.1265,1269-1270.

Disponível em:

www.ti.com/lit/ug/spruhx5e/spruhx5e.pdf

Acesso em 11/12/2017 às 11h16.

[5] SPRS881D – TMS320F2837xS *Delfino Microcontrollers Datasheet.* Ago. 2014, revisada em Jul. 2017. p.97.

Disponível em:

www.ti.com/lit/ds/sprs881d/sprs881d.pdf

Acesso em 11/12/2017 às 14h21.

[6] *Mathworks – PWM Generator.*

Disponível em:

www.mathworks.com/help/physmod/sps/ref/pwmgenerator.html

Acesso em 23/03/2018 às 16h33.

[7] *SGEN Generator Library – Module User's Guide.*
Disponível em:
cfile5.uf.tistory.com/attach/2247CB41580EF7E70665BC
Acesso em 15/06/2018 às 13h19.

Apêndice A

```
#include "F28x_Project.h" // Inclui biblioteca do DSP
void main(void)
{
InitSysCtrl();      // Inicializa o sistema geral
InitGpio();         // Inicialização dos GPIOs
DINT;               // Desabilitar Interrupções globais
InitPieCtrl();      // Inicializar a PIE
IER = 0x0000;       // Desabilita interrupções da CPU
IFR = 0x0000;       // Limpa flags de interrupção
InitPieVectTable(); // Inicializa a tabela da PIE com ponteiros
EINT;               // Habilita interrupções globais
ERTM;
```

Apêndice B

```
#include "F28x_Project.h" // Inclui biblioteca do DSP
void ConfigureADC(void);                  // Função que irá inicializar o ADC
void SetupADCContinuous(Uint16 channel); // Função que irá configurar o ADC
#define RESULTS_BUFFER_SIZE 256          // Tamanho do Vetor de resultados
Uint16 AdcaResults[RESULTS_BUFFER_SIZE]; // Vetor de resultados
Uint16 resultsIndex;                     // Índice do vetor de resultados
void main(void)
{
InitSysCtrl();      // Inicializa o sistema geral
InitGpio();         // Inicialização dos GPIOs
DINT;               // Desabilitar Interrupções globais
InitPieCtrl();      // Inicializar a PIE
IER = 0x0000;       // Desabilita interrupções da CPU
IFR = 0x0000;       // Limpa flags de interrupção
InitPieVectTable(); // Inicializa a tabela da PIE com ponteiros
ConfigureADC();     // Inicialização do ADC
SetupADCContinuous(0); // Canal 0 em conversões contínuas
EINT;                  // Habilita interrupções globais
ERTM;                  // Habilita interrupções de tempo real
// Inicializar o vetor de resultados
for(resultsIndex = 0; resultsIndex < RESULTS_BUFFER_SIZE; resultsIndex++)
{
AdcaResults[resultsIndex] = 0;
}
resultsIndex = 0;
do
{
// Habilitar Flags
AdcaRegs.ADCINTSEL1N2.bit.INT1E = 1;
AdcaRegs.ADCINTSEL1N2.bit.INT2E = 1;
AdcaRegs.ADCINTSEL3N4.bit.INT3E = 1;
AdcaRegs.ADCINTSEL3N4.bit.INT4E = 1;
AdcaRegs.ADCINTFLGCLR.all = 0x000F; // Limpa flags
// Inicializa o índice
resultsIndex = 0;
//Software força SOCs 0 à 7
AdcaRegs.ADCSOCFRC1.all = 0x00FF;
// Continua amostrando até que o vetor de resultados esteja cheio
while(resultsIndex < RESULTS_BUFFER_SIZE)
{
// Espera até que as 8 primeiras conversões sejam completadas
while(0 == AdcaRegs.ADCINTFLG.bit.ADCINT3);
// Limpa ambos os flags de INT pelas primeiras 8 conversões
AdcaRegs.ADCINTFLGCLR.bit.ADCINT1 = 1;
AdcaRegs.ADCINTFLGCLR.bit.ADCINT3 = 1;
// Salva resultados no vetor
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT0;
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT1;
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT2;
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT3;
```

```
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT4;
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT5;
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT6;
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT7;
// Espera pelas próximas 8 conversões
while(0 == AdcaRegs.ADCINTFLG.bit.ADCINT4);
// Limpa ambos os flags de INT pelas 8 conversões decorridas
AdcaRegs.ADCINTFLGCLR.bit.ADCINT2 = 1;
AdcaRegs.ADCINTFLGCLR.bit.ADCINT4 = 1;
// Salva resultados no vetor
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT8;
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT9;
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT10;
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT11;
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT12;
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT13;
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT14;
AdcaResults[resultsIndex++] = AdcaResultRegs.ADCRESULT15;
}
// Desabilita flags para interromper a amostragem
AdcaRegs.ADCINTSEL1N2.bit.INT1E = 0;
AdcaRegs.ADCINTSEL1N2.bit.INT2E = 0;
AdcaRegs.ADCINTSEL3N4.bit.INT3E = 0;
AdcaRegs.ADCINTSEL3N4.bit.INT4E = 0;
asm("   ESTOP0"); // breakpoint de software
}while(1);
}
void ConfigureADC(void)
{
EALLOW;
AdcaRegs.ADCCTL2.bit.PRESCALE = 6; // Divide ADCCLK por 4
// Configura o ADCA para trabalhar em 12-bits e em modo single-ended
AdcSetMode(ADC_ADCA, ADC_RESOLUTION_12BIT, ADC_SIGNALMODE_SINGLE);
// O pulso de trigger da interrupção será dado após o final da conversão
AdcaRegs.ADCCTL1.bit.INTPULSEPOS = 1;
AdcaRegs.ADCCTL1.bit.ADCPWDNZ = 1; // Energiza o ADC
// Delay de 1 ms para garantir inicialização total do ADC
DELAY_US(1000);
EDIS;
}
void SetupADCContinuous(Uint16 channel)
{
Uint16 acqps;
// Define a janela de aquisição mínima em SYSCLKS de acordo com a resolução
if(ADC_RESOLUTION_12BIT == AdcaRegs.ADCCTL2.bit.RESOLUTION){
acqps = 14; // 75 ns
}
else { // Resolução 16 bits
acqps = 63; // 320 ns
}
EALLOW;
AdcaRegs.ADCSOC0CTL.bit.CHSEL  = channel;  //SOC irá converter em channel
AdcaRegs.ADCSOC1CTL.bit.CHSEL  = channel;
AdcaRegs.ADCSOC2CTL.bit.CHSEL  = channel;
AdcaRegs.ADCSOC3CTL.bit.CHSEL  = channel;
AdcaRegs.ADCSOC4CTL.bit.CHSEL  = channel;
AdcaRegs.ADCSOC5CTL.bit.CHSEL  = channel;
AdcaRegs.ADCSOC6CTL.bit.CHSEL  = channel;
AdcaRegs.ADCSOC7CTL.bit.CHSEL  = channel;
```

```
AdcaRegs.ADCSOC8CTL.bit.CHSEL  = channel;
AdcaRegs.ADCSOC9CTL.bit.CHSEL  = channel;
AdcaRegs.ADCSOC10CTL.bit.CHSEL = channel;
AdcaRegs.ADCSOC11CTL.bit.CHSEL = channel;
AdcaRegs.ADCSOC12CTL.bit.CHSEL = channel;
AdcaRegs.ADCSOC13CTL.bit.CHSEL = channel;
AdcaRegs.ADCSOC14CTL.bit.CHSEL = channel;
AdcaRegs.ADCSOC15CTL.bit.CHSEL = channel;
//Janela de amostragem é acqps + 1 ciclo de SYSCLK
AdcaRegs.ADCSOC0CTL.bit.ACQPS  = acqps;
AdcaRegs.ADCSOC1CTL.bit.ACQPS  = acqps;
AdcaRegs.ADCSOC2CTL.bit.ACQPS  = acqps;
AdcaRegs.ADCSOC3CTL.bit.ACQPS  = acqps;
AdcaRegs.ADCSOC4CTL.bit.ACQPS  = acqps;
AdcaRegs.ADCSOC5CTL.bit.ACQPS  = acqps;
AdcaRegs.ADCSOC6CTL.bit.ACQPS  = acqps;
AdcaRegs.ADCSOC7CTL.bit.ACQPS  = acqps;
AdcaRegs.ADCSOC9CTL.bit.ACQPS  = acqps;
AdcaRegs.ADCSOC10CTL.bit.ACQPS = acqps;
AdcaRegs.ADCSOC11CTL.bit.ACQPS = acqps;
AdcaRegs.ADCSOC12CTL.bit.ACQPS = acqps;
AdcaRegs.ADCSOC13CTL.bit.ACQPS = acqps;
AdcaRegs.ADCSOC14CTL.bit.ACQPS = acqps;
AdcaRegs.ADCSOC15CTL.bit.ACQPS = acqps;
AdcaRegs.ADCINTSEL1N2.bit.INT1E = 0; //Desabilita flag de INT1
AdcaRegs.ADCINTSEL1N2.bit.INT2E = 0; //Desabilita flag de INT2
AdcaRegs.ADCINTSEL3N4.bit.INT3E = 0; //Desabilita flag de INT3
AdcaRegs.ADCINTSEL3N4.bit.INT4E = 0; //Desabilita flag de INT4
//Só serão gerados novos pulsos de ADCINT1 quando o flag for limpado
AdcaRegs.ADCINTSEL1N2.bit.INT1CONT = 0;
//Só serão gerados novos pulsos de ADCINT2 quando o flag for limpado
AdcaRegs.ADCINTSEL1N2.bit.INT2CONT = 0;
//Só serão gerados novos pulsos de ADCINT3 quando o flag for limpado
AdcaRegs.ADCINTSEL3N4.bit.INT3CONT = 0;
//Só serão gerados novos pulsos de ADCINT4 quando o flag for limpado
AdcaRegs.ADCINTSEL3N4.bit.INT4CONT = 0;
AdcaRegs.ADCINTSEL1N2.bit.INT1SEL = 6;  //O fim de SOC6  irá setar flag INT1
AdcaRegs.ADCINTSEL1N2.bit.INT2SEL = 14; //O fim de SOC14 irá setar flag INT2
AdcaRegs.ADCINTSEL3N4.bit.INT3SEL = 7;  //O fim de SOC7  irá setar flag INT3
AdcaRegs.ADCINTSEL3N4.bit.INT4SEL = 15; //O fim de SOC15 irá setar flag INT4
//ADCINT2 será o trigger para os 8 primeiros SOCs
AdcaRegs.ADCINTSOCSEL1.bit.SOC0 = 2;
AdcaRegs.ADCINTSOCSEL1.bit.SOC1 = 2;
AdcaRegs.ADCINTSOCSEL1.bit.SOC2 = 2;
AdcaRegs.ADCINTSOCSEL1.bit.SOC3 = 2;
AdcaRegs.ADCINTSOCSEL1.bit.SOC4 = 2;
AdcaRegs.ADCINTSOCSEL1.bit.SOC5 = 2;
AdcaRegs.ADCINTSOCSEL1.bit.SOC6 = 2;
AdcaRegs.ADCINTSOCSEL1.bit.SOC7 = 2;
//ADCINT1 será o trigger para os próximos 8 SOCs
AdcaRegs.ADCINTSOCSEL2.bit.SOC8 = 1;
AdcaRegs.ADCINTSOCSEL2.bit.SOC9 = 1;
AdcaRegs.ADCINTSOCSEL2.bit.SOC10 = 1;
AdcaRegs.ADCINTSOCSEL2.bit.SOC11 = 1;
AdcaRegs.ADCINTSOCSEL2.bit.SOC12 = 1;
AdcaRegs.ADCINTSOCSEL2.bit.SOC13 = 1;
AdcaRegs.ADCINTSOCSEL2.bit.SOC14 = 1;
AdcaRegs.ADCINTSOCSEL2.bit.SOC15 = 1;
}
```

Apêndice C

```
#include "F28x_Project.h"// Inclui biblioteca do DSP
void ConfigureADC(void);      // Função que irá inicializar o ADC
void SetupADCSoftware(void);// Função que irá configurar o ADC
// Variáveis que armazenarão os resultados de conversão
Uint16 AdcaResult0;
Uint16 AdcaResult1;
Uint16 AdcbResult0;
Uint16 AdcbResult1;
void main(void)
{
InitSysCtrl();            // Inicializa o sistema geral
InitGpio();               // Inicialização dos GPIOs
DINT;                     // Desabilitar interrupções globais
InitPieCtrl();            // Inicializar a PIE
IER = 0x0000;             // Desabilita interrupções da CPU
IFR = 0x0000;             // Limpa flags de interrupção
InitPieVectTable();// Inicializa a tabela da PIE com ponteiros
EINT;                     // Habilita interrupções globais
ERTM;                     // Habilita interrupções em tempo real
ConfigureADC();           // Inicialização do ADC
SetupADCSoftware();       // Configurar o ADC
do
{
// Força SOC0 e SOC1 para ADCA e ADCB, inicializando as conversões
AdcaRegs.ADCSOCFRC1.all = 0x0003;
AdcbRegs.ADCSOCFRC1.all = 0x0003;
// Espera término da conversão de ADCA, e logo após limpa o flag
while(AdcaRegs.ADCINTFLG.bit.ADCINT1 == 0);
AdcaRegs.ADCINTFLGCLR.bit.ADCINT1 = 1;
// Espera término da conversão de ADCB, e logo após limpa o flag
while(AdcbRegs.ADCINTFLG.bit.ADCINT1 == 0);
AdcbRegs.ADCINTFLGCLR.bit.ADCINT1 = 1;
// Armazena resultados
AdcaResult0 = AdcaResultRegs.ADCRESULT0;
AdcaResult1 = AdcaResultRegs.ADCRESULT1;
AdcbResult0 = AdcbResultRegs.ADCRESULT0;
AdcbResult1 = AdcbResultRegs.ADCRESULT1;
asm("   ESTOP0"); // Breakpoint
}while(1);
}
void ConfigureADC(void)
{
EALLOW;
AdcaRegs.ADCCTL2.bit.PRESCALE = 6; // Divide ADCCLK por 4
AdcbRegs.ADCCTL2.bit.PRESCALE = 6; // Divide ADCCLK por 4
// Configura o ADCA e ADCB para trabalhar em 12 bits e em single-ended
AdcSetMode(ADC_ADCA, ADC_RESOLUTION_12BIT,ADC_SIGNALMODE_SINGLE);
```

```
AdcSetMode(ADC_ADCB, ADC_RESOLUTION_12BIT, ADC_SIGNALMODE_SINGLE);
// O pulso de trigger da interrupção será dado após o final da conversão
AdcaRegs.ADCCTL1.bit.INTPULSEPOS = 1;
AdcbRegs.ADCCTL1.bit.INTPULSEPOS = 1;
AdcaRegs.ADCCTL1.bit.ADCPWDNZ = 1; // Energiza o ADC
AdcbRegs.ADCCTL1.bit.ADCPWDNZ = 1; // Energiza o ADC
DELAY_US(1000);//Delay de 1ms para garantir inicialização total do ADC
EDIS;
}
void SetupADCSoftware(void)
{
Uint16 acqps;
// Determina janela mínima para amostragem em SYSCLK com base na resolução
if(ADC_RESOLUTION_12BIT == AdcaRegs.ADCCTL2.bit.RESOLUTION){
acqps = 14; // 75 ns
}
else { // Resolução é 16 bits
acqps = 63; // 320 ns
}
// Seleciona os canais para conversão e flag de EOC para o ADCA
EALLOW;
AdcaRegs.ADCSOC0CTL.bit.CHSEL = 0;  // SOC0 irá converter pino A0
AdcaRegs.ADCSOC0CTL.bit.ACQPS = acqps; // Janela de amostragem é acqps+1 ciclos de
SYSCLK
AdcaRegs.ADCSOC1CTL.bit.CHSEL = 1;     // SOC1 irá converter pino A1
AdcaRegs.ADCSOC1CTL.bit.ACQPS = acqps; // Janela de amostragem é acqps+1ciclos de
SYSCLK
AdcaRegs.ADCINTSEL1N2.bit.INT1SEL = 1; // Fim de SOC1 irá setar o flag INT1
AdcaRegs.ADCINTSEL1N2.bit.INT1E = 1;   // Habilita flag de INT1
AdcaRegs.ADCINTFLGCLR.bit.ADCINT1 = 1; // Limpa flag de INT1
// Seleciona os canais para conversão e flag de EOC para o ADCB
AdcbRegs.ADCSOC0CTL.bit.CHSEL = 0;  // SOC0 irá converter pino B0
AdcbRegs.ADCSOC0CTL.bit.ACQPS = acqps; // Janela de amostragem é acqps+1 ciclos de
SYSCLK
AdcbRegs.ADCSOC1CTL.bit.CHSEL = 1;     // SOC1 irá converter pino B1
AdcbRegs.ADCSOC1CTL.bit.ACQPS = acqps; // Janela de amostragem é acqps+1 ciclos de
SYSCLK
AdcbRegs.ADCINTSEL1N2.bit.INT1SEL = 1; // Fim de SOC1 irá setar o flag INT1
AdcbRegs.ADCINTSEL1N2.bit.INT1E = 1;   // Habilita flag de INT1
AdcbRegs.ADCINTFLGCLR.bit.ADCINT1 = 1; // Limpa flag de INT1
}
```

Apêndice D

```
#include "F28x_Project.h"
// Funções utilizadas no programa declaradas anteriormente
void ConfigureADC(void);
void ConfigureEPWM(void);
void SetupADCEpwm(void);
interrupt void adca1_isr(void);
__interrupt void epwm7_isr(void);
#define EPWM7_TIMER_TBPRD 1628
#define EPWM7_MAX_CMPA     814 // Comparador A máximo
#define EPWM7_MIN_CMPA      50 // Comparador A mínimo
#define EPWM7_MAX_CMPB     814 // Comparador B máximo
#define EPWM7_MIN_CMPB      50 // Comparador B mínimo
#define EPWM7_MAX_DB        50 // Dead-Band máxima
#define EPWM7_MIN_DB         0 // Dead-Band mínima
#define DB_UP                1 // Dead-Band crescente
#define DB_DOWN              0 // Dead-Band decrescente
#define RESULTS_BUFFER_SIZE 10 // Tamanho do vetor de conversões AD
// Variáveis globais do programa
Uint16 AdcaResults[RESULTS_BUFFER_SIZE];
Uint16 resultsIndex;
float32 media=0;
Uint16 i=0;
float32 nmedia=0;
volatileUint16 bufferFull;
//===============================================================================
void main(void)
{
InitSysCtrl(); // Comentado.
InitGpio(); // Comentado.
CpuSysRegs.PCLKCR2.bit.EPWM7=1; // Peripheral Clock Gating Registers, habilita o
ePWM7.
InitEPwm7Gpio(); // Comentado.
EALLOW;
GpioCtrlRegs.GPAMUX1.bit.GPIO13 = 0x00;  // Evitar glitch (LED AZUL).
GpioCtrlRegs.GPAGMUX1.bit.GPIO13 = 0x00; // Mux para EPWM7B estar no led azul.
GpioCtrlRegs.GPAMUX1.bit.GPIO13 = 0x01;  // Mux para EPWM7B estar no led azul.
GpioCtrlRegs.GPAMUX1.bit.GPIO12 = 0x00;  // Evitar glitch (LED VERMELHO).
GpioCtrlRegs.GPAGMUX1.bit.GPIO12 = 0x00; // Mux para EPWM7A estar no led vermelho.
GpioCtrlRegs.GPAMUX1.bit.GPIO12 = 0x01;  // Mux para EPWM7A estar no led vermelho.
GpioCtrlRegs.GPAPUD.bit.GPIO13 = 1;      // Desabilita pullup.
GpioCtrlRegs.GPAPUD.bit.GPIO12 = 1;      // Desabilita pullup.
GpioCtrlRegs.GPADIR.bit.GPIO13 = 1;      // Define como saída.
GpioCtrlRegs.GPADIR.bit.GPIO12 = 1;      // Define como saída.
EDIS;
DINT; // Desabilita interrupções.
InitPieCtrl(); // Comentado.
IER = 0x0000; // Desabilita interrupções da CPU.
IFR = 0x0000; // Limpa flags de interrupção.
```

```
InitPieVectTable(); // Comentado.
EALLOW;
PieVectTable.ADCA1_INT = &adca1_isr; // Função para interrupção do ADCA linkada
com a tabela PIE.
PieVectTable.EPWM7_INT = &epwm7_isr; // Função para interrupção do ePWM7 linkada
com a tabela PIE.
EDIS;
ConfigureADC(); // Inicializa o ADCA e o energiza.
EALLOW;
CpuSysRegs.PCLKCR0.bit.TBCLKSYNC = 0; // Mantém a contagem parada dos módulos
ePWM.
EDIS;
ConfigureEPWM(); // Configura o ePWM7
EALLOW;
CpuSysRegs.PCLKCR0.bit.TBCLKSYNC = 1; // Permite a contagem do smódulos ePWM.
EDIS;
SetupADCEpwm(); // Inicializa o ADCA canal 0
IER |= M_INT1; // Habilita o grupo 1 de interrupções, serve para o ADC
IER |= M_INT3; // Habilita o grupo 3 de interrupções, serve para o ePWM
EINT;          // Habilita as interrupções globais.
ERTM;          // Habilita as interrupções de tempo real.
for(resultsIndex = 0; resultsIndex < RESULTS_BUFFER_SIZE; resultsIndex++)
{
AdcaResults[resultsIndex] = 0;
}
resultsIndex = 0; // Zera índice do vetor de resultados
bufferFull = 0;   // Zera flag de vetor cheio
PieCtrlRegs.PIEIER1.bit.INTx1 = 1; // Habilita interrupção do ADCA
PieCtrlRegs.PIEIER3.bit.INTx7 = 1; // Habilita interrupção do ePWM7
EALLOW;
CpuSysRegs.PCLKCR0.bit.TBCLKSYNC = 1; // Permite a contagem dos módulos ePWM.
EDIS;
EPwm7Regs.ETSEL.bit.SOCAEN = 1;  // Habilita o SOCA
EPwm7Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN; // unfreeze, and enter up count
do
{
}while(1);
}
//===============================================================================
// ConfigureADC - Escreve as configurações e inicialização do ADC.
void ConfigureADC(void)
{
EALLOW;
AdcaRegs.ADCCTL2.bit.PRESCALE = 0;   // Divide ADCCLK por 1.
AdcSetMode(ADC_ADCA, ADC_RESOLUTION_12BIT, ADC_SIGNALMODE_SINGLE); // Inicializa o
ADCA com uma resolução de 12 bits no modo de sinal Single Ended.
AdcaRegs.ADCCTL1.bit.INTPULSEPOS = 1;//Faz com que o pulso de interrupção ocorra
ao final de conversão (EOC) do AD.
AdcaRegs.ADCCTL1.bit.ADCPWDNZ = 1; // Energiza por completo o módulo ADCA.
DELAY_US(1000);  Delay de 1 ms para permitir a correta energização do ADC.
EDIS;
}
//===============================================================================
// ConfigureEPWM - Configure EPWM SOC and compare values
void ConfigureEPWM(void)
{
EALLOW;
// Registros de contagem.
```

```
EPwm7Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN; // Seleciona a contagem crescente e
decrescente, ou seja, triangular.
EPwm7Regs.TBCTL.bit.PHSEN = TB_DISABLE;        // Não carrega TBPHS em TBCTL.
EPwm7Regs.TBPHS.bit.TBPHS = 0x0000;            // A fase é setada como 0.
EPwm7Regs.TBCTR = 0x0000;                      // Limpa o contador.
EPwm7Regs.TBCTL.bit.HSPCLKDIV = TB_DIV1;       // Clock ratio to SYSCLKOUT
EPwm7Regs.TBCTL.bit.CLKDIV = TB_DIV1;
// Registros de Delay
EPwm7Regs.DBCTL.bit.OUT_MODE = DB_FULL_ENABLE; // Habilita delay para as bordas de
subida e descida
EPwm7Regs.DBCTL.bit.POLSEL = DB_ACTV_HIC;      // Controla a seleção de
polaridade, inverte o EPWM7B e funciona no complementar ativo em alto
EPwm7Regs.DBCTL.bit.IN_MODE = DBA_ALL;         // Controle de banda morta, a fonte
para ambos os delays de subida e descida é EPWM7A.
EPwm7Regs.DBRED.bit.DBRED = EPWM7_MAX_DB;      // Delay de subida recebe o valor
de EPWM7_MAX_DB
EPwm7Regs.DBFED.bit.DBFED = EPWM7_MAX_DB;      // Delay de descida recebe o valor
de EPWM7_MAX_DB
// Registros de Shadow
EPwm7Regs.CMPCTL.bit.SHDWAMODE = CC_SHADOW;    // Ativa o modo Shadow para o
comparador A
EPwm7Regs.CMPCTL.bit.SHDWBMODE = CC_SHADOW;    // Ativa o modo Shadow para o
comparador B
EPwm7Regs.CMPCTL.bit.LOADAMODE = CC_CTR_PRD;   // Ativa o carregamento do valor de
Shadow A quando o contador atinge a contagem de PRD
EPwm7Regs.CMPCTL.bit.LOADBMODE = CC_CTR_PRD;   // Ativa o carregamento do valor de
Shadow B quando o contador atinge a contagem de PRD
// Registros de Comparação e Ação
EPwm7Regs.CMPA.bit.CMPA = EPWM7_MAX_CMPA;      // Valor de comparação A
EPwm7Regs.CMPB.bit.CMPB = EPWM7_MAX_CMPB;      // Valor de comparação B
EPwm7Regs.AQCTLA.bit.CAU = AQ_CLEAR;// Seta ePWM7A quando TBCTR = CMPA na contagem
crescente
EPwm7Regs.AQCTLA.bit.CAD = AQ_SET;// Limpa ePWM7A quando TBCTR = CMPA na contagem
decrescente
EPwm7Regs.AQCTLB.bit.CAD = AQ_CLEAR;// Seta ePWM7B quando TBCTR = CMPA na contagem
decrescente
EPwm7Regs.AQCTLB.bit.CAU = AQ_SET;           // Limpa ePWM7B quando TBCTR = CMPA
na contagem crescente
// Desta forma, os valores de ePWM7A e ePWM7B são complementares SEMPRE !!
// Registros de interrupção
EPwm7Regs.ETSEL.bit.INTSEL = ET_CTR_PRD;// Interrupção ocorre quando TBCTR = TBPRD
EPwm7Regs.ETSEL.bit.INTEN = 1;          // Habilita a interrupção
EPwm7Regs.ETPS.bit.INTPRD = ET_1ST;// Gera a interrupção sempre no 1º evento
// Registros de SOC
EPwm7Regs.ETSEL.bit.SOCAEN = 0; // Desabilita SOC no grupo A
EPwm7Regs.ETSEL.bit.SOCASEL = 2; // Dispara o SOC quando TBCTR = TBPRD
EPwm7Regs.ETPS.bit.SOCAPRD = 1;  // Gera pulso de SOC no 1º evento
EPwm7Regs.TBPRD = EPWM7_TIMER_TBPRD; / Seta o período com o valor já definido
EPwm7Regs.TBCTL.bit.CTRMODE = 3;               // Congela o contador
EDIS;
}
//===========================================================================
// SetupADCEpwm - Setup ADC EPWM acquisition window
void SetupADCEpwm(void)
{
Uint16 acqps;
if(ADC_RESOLUTION_12BIT == AdcaRegs.ADCCTL2.bit.RESOLUTION) // Testa a resolução,
em bits, da conversão AD, caso seja 12 bits segue o if
//caso contrário, executa o else, com a resolução de 16 bits.
```

```
{
acqps = 14; //75 ns
}
else//resolution is 16-bit
{
acqps = 63; //320 ns
}
EALLOW;
AdcaRegs.ADCSOC0CTL.bit.CHSEL = 0;    // SOC0 irá converter o pino ADCINA0
AdcaRegs.ADCSOC0CTL.bit.ACQPS = acqps;      // Atribui a janela de amostragem o
valor relativo a resolução pré-estabelecida.
AdcaRegs.ADCSOC0CTL.bit.TRIGSEL = 0x11;// Trigger dado em SOCA0 a partir do ePWM7
// Adição de inicialização também do canal 1 de ADCA
AdcaRegs.ADCSOC1CTL.bit.CHSEL = 1;          // SOC1 irá converter o pino ADCINA1
AdcaRegs.ADCSOC1CTL.bit.ACQPS = acqps;      // Atribui a janela de amostragem o
valor relativo a resolução pré-estabelecida.
AdcaRegs.ADCSOC1CTL.bit.TRIGSEL = 0x11;// Trigger dado em SOCA1 a partir do ePWM7
AdcaRegs.ADCINTSEL1N2.bit.INT1SEL = 1;      // O fim de SOC1 irá setar o flag INT1
AdcaRegs.ADCINTSEL1N2.bit.INT1E = 1;        // Habilita o flag de INT1
AdcaRegs.ADCINTFLGCLR.bit.ADCINT1 = 1;// Garante que o flag de INT1 esteja limpo
    EDIS;
}
//===============================================================================
interrupt void adca1_isr(void)
{
media = 0.19875*(AdcaResultRegs.ADCRESULT0 + AdcaResultRegs.ADCRESULT1);
AdcaRegs.ADCINTFLGCLR.bit.ADCINT1 = 1;  // Limpa o flag de interrupção
PieCtrlRegs.PIEACK.all = PIEACK_GROUP1; // Permite que outras interrupções da
natureza do Grupo 1 ocorram novamente.
}
//===============================================================================
__interrupt void epwm7_isr(void)
{
EPwm7Regs.CMPA.bit.CMPA = media;         // Escreve o novo valor de CMPA de acordo
com a última conversão condicionada
EPwm7Regs.ETCLR.bit.INT = 1;             // Limpa o flag de interrupção
PieCtrlRegs.PIEACK.all = PIEACK_GROUP3;  // Permite que outras interrupções da
natureza do Grupo 3 ocorram novamente
}
```

Apêndice E

```
// Includes:
#include "F28x_Project.h"
// Funções utilizadas no programa declaradas anteriormente:
void ConfigureADC(void);
void ConfigureEPWM(void);
void SetupADCEpwm(void);
interrupt void adca1_isr(void);
interrupt void adcb1_isr(void);
__interrupt void epwm2_isr(void);
__interrupt void epwm6_isr(void);
__interrupt void epwm7_isr(void);
// Defines:
#define EPWM_TIMER_TBPRD 1628 // Contagem máxima de 1628 para termos uma
frequência próxima de 15360 Hz.
#define EPWM_MAX_CMPA     814 // Comparador A máximo
#define EPWM_MIN_CMPA      50 // Comparador A mínimo
#define EPWM_MAX_CMPB      90 // Comparador B máximo
#define EPWM_MIN_CMPB      50 // Comparador B mínimo
#define EPWM_MAX_DB        50 // Dead-Band máxima
#define EPWM_MIN_DB         0 // Dead-Band mínima
#define DB_UP                1 // Dead-Band crescente
#define DB_DOWN              0 // Dead-Band decrescente
// Variáveis globais do programa:
float32 media=0;
Uint16 i=0;
//===============================================================================
void main(void)
{
// Inicialização geral para estado default:
InitSysCtrl();
// Inicialização de GPIO para estado default:
InitGpio();
InitEPwm2Gpio();
InitEPwm6Gpio();
InitEPwm7Gpio(); // Inicialização GPIOs de ePWM7
EALLOW;
GpioCtrlRegs.GPAMUX1.bit.GPIO13 = 0x00;  // Evitar glitch (LED AZUL).
GpioCtrlRegs.GPAGMUX1.bit.GPIO13 = 0x00; // Mux para EPWM7B estar no led azul.
GpioCtrlRegs.GPAMUX1.bit.GPIO13 = 0x01;  // Mux para EPWM7B estar no led azul.
GpioCtrlRegs.GPAMUX1.bit.GPIO12 = 0x00;  // Evitar glitch (LED VERMELHO).
GpioCtrlRegs.GPAGMUX1.bit.GPIO12 = 0x00; // Mux para EPWM7A estar no led vermelho.
GpioCtrlRegs.GPAMUX1.bit.GPIO12 = 0x01;  // Mux para EPWM7A estar no led vermelho.
GpioCtrlRegs.GPAMUX1.bit.GPIO3 = 0x00;  // Evitar glitch (LED AZUL).
GpioCtrlRegs.GPAGMUX1.bit.GPIO3 = 0x00; // Mux para EPWM7B estar no led azul.
GpioCtrlRegs.GPAMUX1.bit.GPIO3 = 0x01;  // Mux para EPWM7B estar no led azul.
```

```
GpioCtrlRegs.GPAMUX1.bit.GPIO2 = 0x00;  // Evitar glitch (LED VERMELHO).
GpioCtrlRegs.GPAGMUX1.bit.GPIO2 = 0x00; // Mux para EPWM7A estar no led vermelho.
GpioCtrlRegs.GPAMUX1.bit.GPIO2 = 0x01;  // Mux para EPWM7A estar no led vermelho.
GpioCtrlRegs.GPAMUX1.bit.GPIO10 = 0x00;  // Evitar glitch (LED AZUL).
GpioCtrlRegs.GPAGMUX1.bit.GPIO10 = 0x00; // Mux para EPWM7B estar no led azul.
GpioCtrlRegs.GPAMUX1.bit.GPIO10 = 0x01;  // Mux para EPWM7B estar no led azul.
GpioCtrlRegs.GPAMUX1.bit.GPIO11 = 0x00;  // Evitar glitch (LED VERMELHO).
GpioCtrlRegs.GPAGMUX1.bit.GPIO11 = 0x00; // Mux para EPWM7A estar no led vermelho.
GpioCtrlRegs.GPAMUX1.bit.GPIO11 = 0x01;  // Mux para EPWM7A estar no led vermelho.
GpioCtrlRegs.GPAPUD.bit.GPIO13 = 1;      // Desabilita pullup.
GpioCtrlRegs.GPAPUD.bit.GPIO12 = 1;      // Desabilita pullup.
GpioCtrlRegs.GPADIR.bit.GPIO13 = 1;      // Define como saída.
GpioCtrlRegs.GPADIR.bit.GPIO12 = 1;      // Define como saída.
GpioCtrlRegs.GPAPUD.bit.GPIO3 = 1;      // Desabilita pullup.
GpioCtrlRegs.GPAPUD.bit.GPIO2 = 1;      // Desabilita pullup.
GpioCtrlRegs.GPADIR.bit.GPIO3 = 1;      // Define como saída.
GpioCtrlRegs.GPADIR.bit.GPIO2 = 1;      // Define como saída.
GpioCtrlRegs.GPAPUD.bit.GPIO10 = 1;      // Desabilita pullup.
GpioCtrlRegs.GPAPUD.bit.GPIO11 = 1;      // Desabilita pullup.
GpioCtrlRegs.GPADIR.bit.GPIO10 = 1;      // Define como saída.
GpioCtrlRegs.GPADIR.bit.GPIO11 = 1;      // Define como saída.
EDIS;
DINT; // Desabilita interrupções.
// Inicializar os registros de controle da PIE
InitPieCtrl();
IER = 0x0000; // Desabilita interrupções da CPU.
IFR = 0x0000; // Limpa flags de interrupção.
// Inicialização da tabela PIE
InitPieVectTable();
EALLOW;
PieVectTable.ADCA1_INT = &adca1_isr; // Função para interrupção do ADCA linkar com
a tabela PIE.
PieVectTable.ADCB1_INT = &adcb1_isr;
PieVectTable.EPWM2_INT = &epwm2_isr;
PieVectTable.EPWM6_INT = &epwm6_isr;
PieVectTable.EPWM7_INT = &epwm7_isr; // Função para interrupção do ePWM7 linkar
com a tabela PIE.
EDIS;
ConfigureADC(); // Inicializa o ADC e o energiza.
ConfigureEPWM(); // Configura o Epwm5, 6 e 7
SetupADCEpwm(); // Inicializa o ADC
IER |= M_INT1; // Habilita o grupo 1 de interrupções, serve para o ADC
IER |= M_INT3; // Habilita o grupo 3 de interrupções, serve para o ePWM
EINT;          // Habilita as interrupções globais.
ERTM;          // Habilita as interrupções de tempo real.
PieCtrlRegs.PIEIER1.bit.INTx1 = 1; // Habilita interrupção do ADCA
PieCtrlRegs.PIEIER1.bit.INTx2 = 1; // Habilita interrupção do ADCB
PieCtrlRegs.PIEIER3.bit.INTx2 = 1; // ePWM5
PieCtrlRegs.PIEIER3.bit.INTx5 = 1; // ePWM6
PieCtrlRegs.PIEIER3.bit.INTx7 = 1; // Habilita interrupção do ePWM7
do
{
// Para uma aplicação adenda, escrever o código aqui para loop infinito
}while(1);
}
//==============================================================================
// ConfigureADC - Escreve as configurações e inicialização do ADC.
void ConfigureADC(void)
{
```

```
EALLOW;
AdcaRegs.ADCCTL2.bit.PRESCALE = 0;// Divide ADCCLK por 1.
AdcSetMode(ADC_ADCA, ADC_RESOLUTION_12BIT, ADC_SIGNALMODE_SINGLE); // Inicializa o
ADCA com uma resolução de 12 bits no modo de sinal Single Ended.
AdcaRegs.ADCCTL1.bit.INTPULSEPOS = 1;// Faz com que o pulso de interrupção ocorra
ao final de conversão (EOC) do AD.
AdcaRegs.ADCCTL1.bit.ADCPWDNZ = 1;// Energiza por completo o módulo ADCA.
DELAY_US(1000);// Delay de 1 ms para permitir a correta energização do ADC.
AdcbRegs.ADCCTL2.bit.PRESCALE = 0;// Divide ADCCLK por 1.
AdcSetMode(ADC_ADCB, ADC_RESOLUTION_12BIT, ADC_SIGNALMODE_SINGLE); // Inicializa o
ADCB com uma resolução de 12 bits no modo de sinal Single Ended.
AdcbRegs.ADCCTL1.bit.INTPULSEPOS = 1;// Faz com que o pulso de interrupção ocorra
ao final de conversão (EOC) do AD.
AdcbRegs.ADCCTL1.bit.ADCPWDNZ = 1;// Energiza por completo o módulo ADCB.
DELAY_US(1000);// Delay de 1 ms para permitir a correta energização do ADC.
EDIS;
}
//===============================================================================
// ConfigureEPWM - Configure EPWM SOC and compare values
void ConfigureEPWM(void)
{
EALLOW;
CpuSysRegs.PCLKCR0.bit.TBCLKSYNC = 0;// Mantém a contagem parada dos módulos ePWM.
// Registros de contagem.
EPwm2Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN; // Seleciona a contagem crescente e
decrescente, ou seja, triangular.
EPwm2Regs.TBCTL.bit.PHSEN = TB_DISABLE;        // Não carrega TBPHS em TBCTL.
EPwm2Regs.TBPHS.bit.TBPHS = 0x0000;            // A fase é setada como 0.
EPwm2Regs.TBCTR = 0x0000;                      // Limpa o contador de contagem.
EPwm2Regs.TBCTL.bit.HSPCLKDIV = TB_DIV1;       // Clock ratio to SYSCLKOUT
EPwm2Regs.TBCTL.bit.CLKDIV = TB_DIV1;
EPwm6Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN; // Seleciona a contagem crescente e
decrescente, ou seja, triangular.
EPwm6Regs.TBCTL.bit.PHSEN = TB_DISABLE;        // Não carrega TBPHS em TBCTL.
EPwm6Regs.TBPHS.bit.TBPHS = 0x0000;            // A fase é setada como 0.
EPwm6Regs.TBCTR = 0x0000;                      // Limpa o contador de contagem.
EPwm6Regs.TBCTL.bit.HSPCLKDIV = TB_DIV1;       // Clock ratio to SYSCLKOUT
EPwm6Regs.TBCTL.bit.CLKDIV = TB_DIV1;
EPwm7Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN; // Seleciona a contagem crescente e
decrescente, ou seja, triangular.
EPwm7Regs.TBCTL.bit.PHSEN = TB_DISABLE;        // Não carrega TBPHS em TBCTL.
EPwm7Regs.TBPHS.bit.TBPHS = 0x0000;            // A fase é setada como 0.
EPwm7Regs.TBCTR = 0x0000;                      // Limpa o contador de contagem.
EPwm7Regs.TBCTL.bit.HSPCLKDIV = TB_DIV1;       // Clock ratio to SYSCLKOUT
EPwm7Regs.TBCTL.bit.CLKDIV = TB_DIV1;
// Registros de Delay
EPwm2Regs.DBCTL.bit.OUT_MODE = DB_FULL_ENABLE; // Habilita delay para as bordas de
subida e descida
EPwm2Regs.DBCTL.bit.POLSEL = DB_ACTV_HIC;      // Controla a seleção de
polaridade, inverte o EPWM2B e funciona no complementar ativo em alto
EPwm2Regs.DBCTL.bit.IN_MODE = DBA_ALL;         // Controle de banda morta, a fonte
para ambos os delays de subida e descida é EPWM2A.
EPwm2Regs.DBRED.bit.DBRED = EPWM_MAX_DB;// Delay de subida recebe o valor de
EPWM_MAX_DB
EPwm2Regs.DBFED.bit.DBFED = EPWM_MAX_DB;// Delay de descida recebe o valor de
EPWM_MAX_DB
EPwm6Regs.DBCTL.bit.OUT_MODE = DB_FULL_ENABLE; // Habilita delay para as bordas de
subida e descida
```

```
EPwm6Regs.DBCTL.bit.POLSEL = DB_ACTV_HIC;      // Controla a seleção de
polaridade, inverte o EPWM6B e funciona no complementar ativo em alto
EPwm6Regs.DBCTL.bit.IN_MODE = DBA_ALL;         // Controle de banda morta, a fonte
para ambos os delays de subida e descida é EPWM6A.
EPwm6Regs.DBRED.bit.DBRED = EPWM_MAX_DB;// Delay de subida recebe o valor de
EPWM_MAX_DB
EPwm6Regs.DBFED.bit.DBFED = EPWM_MAX_DB;// Delay de descida recebe o valor de
EPWM_MAX_DB
EPwm7Regs.DBCTL.bit.OUT_MODE = DB_FULL_ENABLE; // Habilita delay para as bordas de
subida e descida
EPwm7Regs.DBCTL.bit.POLSEL = DB_ACTV_HIC;      // Controla a seleção de
polaridade, inverte o EPWM7B e funciona no complementar ativo em alto
EPwm7Regs.DBCTL.bit.IN_MODE = DBA_ALL;         // Controle de banda morta, a fonte
para ambos os delays de subida e descida é EPWM7A.
EPwm7Regs.DBRED.bit.DBRED = EPWM_MAX_DB;// Delay de subida recebe o valor de
EPWM7_MAX_DB
EPwm7Regs.DBFED.bit.DBFED = EPWM_MAX_DB;// Delay de descida recebe o valor de
EPWM7_MAX_DB
// Registros de Shadow
EPwm2Regs.CMPCTL.bit.SHDWAMODE = CC_SHADOW;    // Ativa o modo Shadow para o
comparador A
EPwm2Regs.CMPCTL.bit.SHDWBMODE = CC_SHADOW;    // Ativa o modo Shadow para o
comparador B
EPwm2Regs.CMPCTL.bit.LOADAMODE = CC_CTR_PRD;   // Ativa o carregamento do valor de
Shadow A quando o contador atinge a contagem de PRD
EPwm2Regs.CMPCTL.bit.LOADBMODE = CC_CTR_PRD;   // Ativa o carregamento do valor de
Shadow B quando o contador atinge a contagem de PRD
EPwm6Regs.CMPCTL.bit.SHDWAMODE = CC_SHADOW;    // Ativa o modo Shadow para o
comparador A
EPwm6Regs.CMPCTL.bit.SHDWBMODE = CC_SHADOW;    // Ativa o modo Shadow para o
comparador B
EPwm6Regs.CMPCTL.bit.LOADAMODE = CC_CTR_PRD;   // Ativa o carregamento do valor de
Shadow A quando o contador atinge a contagem de PRD
EPwm6Regs.CMPCTL.bit.LOADBMODE = CC_CTR_PRD;   // Ativa o carregamento do valor de
Shadow B quando o contador atinge a contagem de PRD
EPwm7Regs.CMPCTL.bit.SHDWAMODE = CC_SHADOW;    // Ativa o modo Shadow para o
comparador A
EPwm7Regs.CMPCTL.bit.SHDWBMODE = CC_SHADOW;    // Ativa o modo Shadow para o
comparador B
EPwm7Regs.CMPCTL.bit.LOADAMODE = CC_CTR_PRD;   // Ativa o carregamento do valor de
Shadow A quando o contador atinge a contagem de PRD
EPwm7Regs.CMPCTL.bit.LOADBMODE = CC_CTR_PRD;   // Ativa o carregamento do valor de
Shadow B quando o contador atinge a contagem de PRD
// Registros de Comparação e Ação
EPwm2Regs.CMPA.bit.CMPA = EPWM_MAX_CMPA;      // Valor de comparação A
EPwm2Regs.CMPB.bit.CMPB = EPWM_MAX_CMPB;      // Valor de comparação B
EPwm2Regs.AQCTLA.bit.CAU = AQ_CLEAR;// Seta ePWM7A quando TBCTR = CMPA na contagem
crescente
EPwm2Regs.AQCTLA.bit.CAD = AQ_SET;// Limpa ePWM2A quando TBCTR = CMPA na contagem
decrescente
EPwm2Regs.AQCTLB.bit.CAD = AQ_CLEAR;// Seta ePWM2B quando TBCTR = CMPA na contagem
decrescente
EPwm2Regs.AQCTLB.bit.CAU = AQ_SET;// Limpa ePWM2B quando TBCTR = CMPA na contagem
crescente
EPwm6Regs.CMPA.bit.CMPA = EPWM_MAX_CMPA;      // Valor de comparação A
EPwm6Regs.CMPB.bit.CMPB = EPWM_MAX_CMPB;      // Valor de comparação B
EPwm6Regs.AQCTLA.bit.CAU = AQ_CLEAR;// Seta ePWM7A quando TBCTR = CMPA na contagem
crescente
```

```
EPwm6Regs.AQCTLA.bit.CAD = AQ_SET;// Limpa ePWM6A quando TBCTR = CMPA na contagem
decrescente
EPwm6Regs.AQCTLB.bit.CAD = AQ_CLEAR;// Seta ePWM6B quando TBCTR = CMPA na contagem
decrescente
EPwm6Regs.AQCTLB.bit.CAU = AQ_SET;// Limpa ePWM6B quando TBCTR = CMPA na contagem
crescente
EPwm7Regs.CMPA.bit.CMPA = EPWM_MAX_CMPA;      // Valor de comparação A
EPwm7Regs.CMPB.bit.CMPB = EPWM_MAX_CMPB;      // Valor de comparação B
EPwm7Regs.AQCTLA.bit.CAU = AQ_CLEAR;// Seta ePWM7A quando TBCTR = CMPA na contagem
crescente
EPwm7Regs.AQCTLA.bit.CAD = AQ_SET;// Limpa ePWM7A quando TBCTR = CMPA na contagem
decrescente
EPwm7Regs.AQCTLB.bit.CAD = AQ_CLEAR;// Seta ePWM7B quando TBCTR = CMPA na contagem
decrescente
EPwm7Regs.AQCTLB.bit.CAU = AQ_SET;// Limpa ePWM7B quando TBCTR = CMPA na contagem
crescente
// Desta forma, os valores de ePWMA e ePWMB são complementares SEMPRE !!
// Registros de interrupção
EPwm2Regs.ETSEL.bit.INTSEL = ET_CTR_PRD;// Interrupção ocorre quando TBCTR = TBPRD
EPwm2Regs.ETSEL.bit.INTEN = 1;// Habilita a interrupção
EPwm2Regs.ETPS.bit.INTPRD = ET_1ST;// Gera a interrupção sempre no 1º evento
EPwm6Regs.ETSEL.bit.INTSEL = ET_CTR_PRD;// Interrupção ocorre quando TBCTR = TBPRD
EPwm6Regs.ETSEL.bit.INTEN = 1;// Habilita a interrupção
EPwm6Regs.ETPS.bit.INTPRD = ET_1ST;// Gera a interrupção sempre no 1º evento
EPwm7Regs.ETSEL.bit.INTSEL = ET_CTR_PRD;// Interrupção ocorre quando TBCTR = TBPRD
EPwm7Regs.ETSEL.bit.INTEN = 1;// Habilita a interrupção
EPwm7Regs.ETPS.bit.INTPRD = ET_1ST;// Gera a interrupção sempre no 1º evento
// Registros de SOC
EPwm2Regs.ETSEL.bit.SOCAEN = 0;// Desabilita SOC no grupo A
EPwm2Regs.ETSEL.bit.SOCASEL = 2;// Dispara o SOC quando TBCTR = TBPRD
EPwm2Regs.ETPS.bit.SOCAPRD = 1;// Gera pulso de SOC no 1º evento
EPwm2Regs.TBPRD = EPWM_TIMER_TBPRD;// Seta o período com o valor já definido
EPwm2Regs.TBCTL.bit.CTRMODE = 3;// Congela o contador
EPwm6Regs.ETSEL.bit.SOCAEN = 0;// Desabilita SOC no grupo A
EPwm6Regs.ETSEL.bit.SOCASEL = 2;// Dispara o SOC quando TBCTR = TBPRD
EPwm6Regs.ETPS.bit.SOCAPRD = 1;// Gera pulso de SOC no 1º evento
EPwm6Regs.TBPRD = EPWM_TIMER_TBPRD;// Seta o período com o valor já definido
EPwm6Regs.TBCTL.bit.CTRMODE = 3;// Congela o contador
EPwm7Regs.ETSEL.bit.SOCAEN = 0;// Desabilita SOC no grupo A
EPwm7Regs.ETSEL.bit.SOCASEL = 2;// Dispara o SOC quando TBCTR = TBPRD
EPwm7Regs.ETPS.bit.SOCAPRD = 1;// Gera pulso de SOC no 1º evento
EPwm7Regs.TBPRD = EPWM_TIMER_TBPRD;// Seta o período com o valor já definido
EPwm7Regs.TBCTL.bit.CTRMODE = 3;// Congela o contador
CpuSysRegs.PCLKCR0.bit.TBCLKSYNC = 1;// Permite a contagem dos módulos ePWM.
EPwm2Regs.ETSEL.bit.SOCAEN = 1;               // Habilita o SOCA
EPwm2Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN; // Descongela a contagem
EPwm6Regs.ETSEL.bit.SOCAEN = 1;               // Habilita o SOCA
EPwm6Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN; // Descongela a contagem
EPwm7Regs.ETSEL.bit.SOCAEN = 1;               // Habilita o SOCA
EPwm7Regs.TBCTL.bit.CTRMODE = TB_COUNT_UPDOWN; // Descongela a contagem
EDIS;
}
//===========================================================================
// SetupADCEpwm - Setup ADC EPWM acquisition window
void SetupADCEpwm(void)
{
Uint16 acqpsA, acqpsB;
```

```
if(ADC_RESOLUTION_12BIT == AdcaRegs.ADCCTL2.bit.RESOLUTION) // Testa a resolução,
em bits, da conversão AD, caso seja 12 bits segue o if caso contrário, executa o
else, com a resolução de 16 bits.
{
acqpsA = 14; //75 ns
}
else //resolution is 16-bit
{
acqpsA = 63; //320 ns
}
EALLOW;
AdcaRegs.ADCSOC0CTL.bit.CHSEL = 0;         // SOC0 irá converter o pino ADCINA0
AdcaRegs.ADCSOC0CTL.bit.ACQPS = acqpsA;      // Atribui a janela de amostragem o
valor relativo a resolução pré-estabelecida.
AdcaRegs.ADCSOC0CTL.bit.TRIGSEL = 0x11;// Trigger dado em SOCA0 a partir do ePWM7
// Adição de inicialização também do canal 1 de ADCA
AdcaRegs.ADCSOC1CTL.bit.CHSEL = 1;          // SOC1 irá converter o pino ADCINA1
AdcaRegs.ADCSOC1CTL.bit.ACQPS = acqpsA;      // Atribui a janela de amostragem o
valor relativo a resolução pré-estabelecida.
AdcaRegs.ADCSOC1CTL.bit.TRIGSEL = 0x11;// Trigger dado em SOCA1 a partir do ePWM7
AdcaRegs.ADCINTSEL1N2.bit.INT1SEL = 1;      // O fim de SOC1 irá setar o flag INT1
AdcaRegs.ADCINTSEL1N2.bit.INT1E = 1;        // Habilita o flag de INT1
AdcaRegs.ADCINTFLGCLR.bit.ADCINT1 = 1;// Garante que o flag de INT1 esteja limpo
if(ADC_RESOLUTION_12BIT == AdcbRegs.ADCCTL2.bit.RESOLUTION) // Testa a resolução,
em bits, da conversão AD, caso seja 12 bits segue o if caso contrário, executa o
else, com a resolução de 16 bits.
{
acqpsB = 14; //75 ns
}
else //resolution is 16-bit
{
acqpsB = 63; //320 ns
}
AdcbRegs.ADCSOC0CTL.bit.CHSEL = 0;
AdcbRegs.ADCSOC0CTL.bit.ACQPS = acqpsB;      // Atribui a janela de amostragem o
valor relativo a resolução pré-estabelecida.
AdcbRegs.ADCSOC0CTL.bit.TRIGSEL = 0x11;// Trigger dado em SOCB0 a partir do ePWM7
// Adição de inicialização também do canal 1 de ADCB
AdcbRegs.ADCSOC1CTL.bit.CHSEL = 1;          // SOC1 irá converter o pino ADCINB1
AdcbRegs.ADCSOC1CTL.bit.ACQPS = acqpsB;      // Atribui a janela de amostragem o
valor relativo a resolução pré-estabelecida.
AdcbRegs.ADCSOC1CTL.bit.TRIGSEL = 0x11;// Trigger dado em SOCB1 a partir do ePWM7
AdcbRegs.ADCINTSEL1N2.bit.INT1SEL = 1;      // O fim de SOC1 irá setar o flag INT1
AdcbRegs.ADCINTSEL1N2.bit.INT1E = 1;        // Habilita o flag de INT1
AdcbRegs.ADCINTFLGCLR.bit.ADCINT1 = 1;// Garante que o flag de INT1 esteja limpo
EDIS;
}
//==============================================================================
interrupt void adca1_isr(void)
{
media = 0.19875*(AdcaResultRegs.ADCRESULT0 + AdcaResultRegs.ADCRESULT1);
AdcaRegs.ADCINTFLGCLR.bit.ADCINT1 = 1;  // Limpa o flag de interrupção
PieCtrlRegs.PIEACK.all = PIEACK_GROUP1; // Permite que outras interrupções da
natureza do Grupo 1 ocorram novamente
}
//==============================================================================
==================================================================
interrupt void adcb1_isr(void)
{
```

```
}
//==============================================================================
__interrupt void epwm2_isr(void)
{
EPwm2Regs.ETCLR.bit.INT = 1;             // Limpa o flag de interrupção
PieCtrlRegs.PIEACK.all = PIEACK_GROUP3;  // Permite que outras interrupções da
natureza do Grupo 3 ocorram novamente
}
//==============================================================================
__interrupt void epwm6_isr(void)
{
EPwm6Regs.ETCLR.bit.INT = 1;             // Limpa o flag de interrupção
PieCtrlRegs.PIEACK.all = PIEACK_GROUP3;  // Permite que outras interrupções da
natureza do Grupo 3 ocorram novamente
}
//==============================================================================
__interrupt void epwm7_isr(void)
{
EPwm7Regs.CMPA.bit.CMPA = media;// Escreve o novo valor de CMPA de acordo com a
última conversão condicionada
EPwm7Regs.ETCLR.bit.INT = 1;             // Limpa o flag de interrupção
PieCtrlRegs.PIEACK.all = PIEACK_GROUP3;  // Permite que outras interrupções da
natureza do Grupo 3 ocorram novamente
}
```

Apêndice F

```
// Inclusão de bibliotecas
#include "F28x_Project.h"
// Variáveis globais
Volatile struct DAC_REGS* DAC_PTR[4] = {0x0,&DacaRegs,&DacbRegs,&DaccRegs};
Uint16 dacval = 4096;
// Defines
#define REFERENCE_VDAC      0
#define REFERENCE_VREF      1
#define DACA                1
#define DACB                2
#define DACC                3
#define REFERENCE           REFERENCE_VREF
#define DAC_NUM             DACA
// Funções protótipo
void configureDAC(Uint16 dac_num);
void main(void)
{
InitSysCtrl();
DINT;
InitPieCtrl();
IER = 0x0000;
IFR = 0x0000;
InitPieVectTable();
configureDAC(DAC_NUM);
while(1)
{
DAC_PTR[DAC_NUM]->DACVALS.all = dacval;
//DacaRegs.DACVALS.all = dacval;
DELAY_US(2);
}
}
// configureDAC - Configure specified DAC output
void configureDAC(Uint16 dac_num)
{
EALLOW;
//DacaRegs.DACCTL.bit.DACREFSEL = REFERENCE;
//DacaRegs.DACOUTEN.bit.DACOUTEN = 1;
//DacaRegs.DACVALS.all = 0;
DAC_PTR[dac_num]->DACCTL.bit.DACREFSEL = REFERENCE;
DAC_PTR[dac_num]->DACOUTEN.bit.DACOUTEN = 1;
DAC_PTR[dac_num]->DACVALS.all = 0;
DELAY_US(10); // Delay for buffered DAC to power up
EDIS;
```

}

Apêndice G

```
%% Criação portadora ePWM contagem decrescente.
% Data: 26/03/2018.
% Feito por: Felipe Bordoni Diniz.
%-------------------------------------------------------------------------
% Contagem máxima estipulada em 2500,
% passo de contagem 0.01 s ou 100 Hz,
% Período da portadora 0.4 Hz,
% Serão mostrados dois períodos de contagem desta onda ePWM portadora.
%-------------------------------------------------------------------------
close all
clear all
clc
x = (0:0.01:5); % Vetor de valores no tempo
a1 = 2500; % Talor máximo de contagem
T = 2.5; % Período da onda portadora em segundos
y = zeros(1,length(x)); % Vetor de valores de contagem
for i=1:(floor(length(x)/2))
    y(i) = -a1.*x(i)./T + a1;
    y(i+floor(length(x)/2)) = y(i);
end
figure(1), plot(x,y,'LineWidth',4.0), grid on;
figure(1), title('ePWM contagem decrescente'), xlabel('Tempo [s]'),
ylabel('Contagem do ePWM');
%% Criação portadora ePWM contagem crescente e decrescente.
% Data: 26/03/2018.
% Feito por: Felipe Bordoni Diniz.
%-------------------------------------------------------------------------
% Contagem máxima estipulada em 2500,
% passo de contagem 0.01 s ou 100 Hz,
% Período da portadora 0.4 Hz,
% Serão mostrados dois períodos de contagem desta onda ePWM portadora.
%-------------------------------------------------------------------------
clear all
clc
x = (0:0.01:5); % Vetor de valores no tempo
a1 = 2500; % Talor máximo de contagem
T = 2.5; % Período da onda portadora em segundos
y = zeros(1,length(x)); % Vetor de valores de contagem
for i=1:(floor(length(x)/4))
    y(i) = 2*a1.*x(i)./T;
    y(i+floor(length(x)/2)) = y(i);
end
```

```
for i=(floor(length(x)/4)+1):floor(length(x)/2)
    y(i) = -2*a1.*x(i)./T + 2*a1;
    y(i+floor(length(x)/2)) = y(i);
end
figure(2), plot(x,y,'LineWidth',4.0), grid on;
figure(2), title('ePWM contagem crescente e decrescente'), xlabel('Tempo
[s]'), ylabel('Contagem do ePWM');
%% Simulação de Tempo-Morto no ePWM
% Data: 02/04/2018
% Feito por: Felipe Bordoni Diniz
%-------------------------------------------------------------------------
% Esta simulação apresenta o efeito do tempo-morto em duas ondas
% complementares de ePWM.
% Setado para tempo-morto de 3 segundos e período de 40 segundos
%-------------------------------------------------------------------------
clc % Limpar tela de comandos
clear all% Limpar Workspace
highA=ones(1,20); % Vetor de 1's
lowA=zeros(1,20); % Vetor de 0's
PWMA=horzcat(highA,lowA); % Concatenação de 1 e 0
for i=1:1
    PWMA=horzcat(PWMA,highA,lowA); % Concatenar i vezes
end
PWMB=double(~PWMA); % PWMB é inverso à PWMA
db=3; % Definição do tempo-morto
T=length(PWMA)/(length(highA)+length(lowA)); % Número de períodos
for j=1:T
for i=1:db
PWMB(length(highA)*(2*j-1)+i)=0; % Aplicar tempo-morto em PWMB
        PWMA((length(highA)+length(lowA))*(j-1)+i)=0; % Aplicar tempo-morto
em PWMA
end
end
% Plotando as curvas na mesma Figura
figure(3), plot(PWMA, 'LineWidth', 4.0);
figure(3), hold on;
figure(3), grid on;
figure(3), plot(PWMB, 'r', 'LineWidth', 4.0);
figure(3), title('Saídas ePWM com Tempo-Morto');
figure(3), xlabel('tempo [s]');
figure(3), ylabel('Nível lógico');
figure(3), legend('ePWM A','ePWM B');
%% Variando o ciclo de trabalho da saida ePWM
% Data: 16/04/2018
% Feito por: Felipe Bordoni Diniz
%-------------------------------------------------------------------------
% Explicar
%-------------------------------------------------------------------------
clc % Limpar tela de comandos
clear all% Limpar Workspace
clear all
clc
x = (0:0.01:5); % Vetor de valores no tempo
a1 = 2500; % Talor máximo de contagem
T = 2.5; % Período da onda portadora em segundos
y = zeros(1,length(x)); % Vetor de valores de contagem
dutyA=0.2*a1;
dutyB=0.8*a1;
A=zeros(1,length(x));
B=zeros(1,length(x));
for i=1:(floor(length(x)/4))
```

```
    y(i) = 2*a1.*x(i)./T;
    y(i+floor(length(x)/2)) = y(i);
end
for i=(floor(length(x)/4)+1):floor(length(x)/2)
    y(i) = -2*a1.*x(i)./T + 2*a1;
    y(i+floor(length(x)/2)) = y(i);
end
for i=1:length(x)
if dutyA>y(i)||dutyA==y(i)
        A(i)=1;
end
end
for i=1:length(x)
if dutyB>y(i)||dutyB==y(i)
        B(i)=1;
end
end
figure(4), subplot(2,2,1), plot(x,y,'LineWidth',4.0,'Color','b'), grid on;
figure(4), subplot(2,2,1), title('ePWM contagem crescente e decrescente'),
xlabel('Tempo [s]'), ylabel('Contagem do ePWM');
figure(4), subplot(2,2,1), hold on;
figure(4), subplot(2,2,1), plot(x,dutyA,'Color','r');
figure(4), subplot(2,2,1), axis([x(1) x(length(x)) 0 a1]);
figure(4), subplot(2,2,3), plot(x,A,'LineWidth',4.0,'Color','k'), grid on;
figure(4), subplot(2,2,3), axis([x(1) x(length(x)) 0 1]);
figure(4), subplot(2,2,3), title('Saída ePWM A'), xlabel('Tempo [s]'),
ylabel('Nível lógico');
figure(4), subplot(2,2,2), plot(x,y,'LineWidth',4.0,'Color','b'), grid on;
figure(4), subplot(2,2,2), title('ePWM contagem crescente e decrescente'),
xlabel('Tempo [s]'), ylabel('Contagem do ePWM');
figure(4), subplot(2,2,2), hold on;
figure(4), subplot(2,2,2), plot(x,dutyB,'Color','r');
figure(4), subplot(2,2,2), axis([x(1) x(length(x)) 0 a1]);
figure(4), subplot(2,2,4), plot(x,B,'LineWidth',4.0,'Color','k'), grid on;
figure(4), subplot(2,2,4), axis([x(1) x(length(x)) 0 1]);
figure(4), subplot(2,2,4), title('Saída ePWM A'), xlabel('Tempo [s]'),
ylabel('Nível lógico');
```